Universidade de Aveiro
Departamento de Eletrónica, Telecomunicações e Informática
2016

Manuel Fernandes

A Rede como um serviço usando o Openstack - Neutron

Network as a Service using Openstack - Neutron

Manuel Fernandes

A Rede como um serviço usando o Openstack - Neutron

Network as a Service using Openstack - Neutron

"*Ambition is the path to success. Persistence is the vehicle you arrive in.*"

— Bill Bradley

Universidade de Aveiro
2016

Departamento de Eletrónica,
Telecomunicações e Informática

Manuel Fernandes

A Rede como um serviço usando o Openstack - Neutron

Network as a Service using Openstack - Neutron

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Doutor Diogo Nuno Pereira Gomes, Professor auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e do Doutor Rui Luís Andrade Aguiar, Professor catedrático do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro.

o júri / the jury

presidente / president

Prof. Doutora Susana Isabel Barreto de Miranda Sargento
Professora Associada C/ Agregação da Universidade de Aveiro

vogais / examiners committee

Doutor Francisco Manuel Marques Fontes
Consultor Sénior da PT Inovação e Sistemas

Prof. Doutor Diogo Nuno Pereira Gomes
Professor Auxiliar da Universidade de Aveiro

agradecimentos /
acknowledgements

Agradeço aos orientadores desta dissertação e ao grupo de trabalho de OpenStack do ATNoG. Agradeço também com relevo todo o importante apoio e incentivo da minha família, amigos e colegas e ao longo deste meu percurso académico.

Palavras Chave

computação na nuvem, redes heterogéneas, redes legadas, virtualização de redes, redes definidas por software.

Resumo

Hoje em dia as necessidades ao nível de recursos computacionais levam, de uma forma geral, a um aumento do uso de técnicas cada vez mais aprimoradas para rentabilizar a sua utilização. Isto acontece devido a um número cada vez maior de serviços baseados em tecnologias da informação. O paradigma da computação Cloud foi uma das respostas dadas a essas necessidades. Com ele, cresce igualmente a necessidade de procura por novas soluções ao nível das redes de telecomunicações, para responder de uma maneira pronta e capaz a esta nova realidade.

Para muitas empresas, de forma a para manterem a sua competitividade, o custo de modernizar as redes já existentes na sua infraestrutura para uma solução baseada em Cloud, envolve um esforço financeiro considerável, que numa grande maioria dos casos não é rentável. Pretende-se com esta dissertação apresentar uma possível solução para este problema, através do desenvolvimento de uma framework que tem como objetivo dotar os softwares de gestão de Cloud, de capacidades para criar uma ponte entre as redes geridas por estes, e a infraestrutura de rede já existente na empresa. Este objetivo será atingido configurando de forma automática os dispositivos de rede existentes na infraestrutura, de uma forma não disruptiva. Como plataforma de teste, o software de Cloud escolhido foi o OpenStack, mais precisamente o seu componente de gestão de rede chamado Neutron. A este foram adicionadas novas APIs e novas entidades à base de dados para suportar a nova funcionalidade.

Keywords

Cloud computing, heterogeneous networks, legacy networks, network virtualization, software defined networks.

Abstract

Nowadays the needs regarding of Computational resources, usually lead, to an increase in the use of improved techniques to monetize its usage. This happens due to an increasing number of services based on Information technologies. The Cloud computing paradigm was one of the awnsers given to these needs, and, with it, there is also a growing need for the search for new solutions in telecommunications networks to respond in a ready and capable way to this new reality.

For many companies, in order to remain competitive, the cost of modernizing their existing infrastructure networks for a cloud-based solution involves a considerable financial effort that, in most cases, is not profitable. This dissertation intends to present a possible solution to this problem, through the development of a framework that aims to provide Cloud management software, with the ability to create a bridge between the networks they manage, and the network infrastructure already existing in the company. This goal will be achieved by automatically configuring the existing network devices in the infrastructure in a non-disruptive way. As a test platform, the chosen Cloud software was OpenStack, more precisely its network management component called Neutron. New APIs and new database entities have been added to Neutron to support the new functionality.

Conteúdo

Lista de Figuras

Lista de Tabelas

Glossário

IT	*Information Technologies*
NIST	*National Institute of Standards and Technology*
SOA	*Service-Oriented Architecture*
SDN	*Software-Defined Networking*
NFV	*Network Functions Virtualization*
VLAN	*Virtual Local Area Network*
GRE	*Generic Routing Encapsulation*
VXLAN	*Virtual Extensible LAN*
VTEP	*VXLAN Tunnel End Point*
VPN	*Virtual Private Network*
LAN	*Local Area Network*
SNMP	*Simple Network Management Protocol*
ICMP	*Internet Control Message Protocol*
MAC	*Media Access Control*
GPU	*Graphics Processing Unit*
CPU	*Central Processing Unit*
API	*Application Programming Interface*
FOSS	*Free and Open-Source Software*
GUI	*Graphical User Interface*
CLI	*Command Line Interface*
SaaS	*Software as a Service*
PaaS	*Platform as a Service*
IaaS	*Infrastructure as a Service*
NaaS	*Network as a Service*
FaaS	*Firewall as a Service*
OVS	Open vSwitch
OVSDB	Open vSwitch Database
IoT	*Internet of Things*
VM	*Virtual Machine*
OS	*Operating System*
SAN	*Storage Area Network*
NFS	*Network File System*
SSH	*Secure Shell*
IPC	*Inter-Process Communication*
AP	*Access Point*
SSID	*Service Set Identifier*
L2-GW	*Layer-2 Gateway*
TELCO	*Telecommunications Company*
ML2	*Modular Layer-2*
L2	*Layer-2*
L3	*Layer-3*
WAN	*Wide Area Network*
vSwitch	*Virtual Switch*
CAPEX	*Capital Expenditure*
OPEX	*Operational Expenditure*
CMS	*Cloud Management Software*
STIC	Serviços de Tecnologias de Informação e Comunicação
CRUD	*Create, Read, Update and Delete*
RPC	*Remote Procedure Call*
KVM	*Kernel-based Virtual Machine*
NAT	*Network Address Translation*
SNAT	*Stateful NAT*
DHCP	*Dynamic Host Configuration Protocol*

LDAP	*Lightweight Directory Access Protocol*
ORM	*Object-Relational Mapping*
EXTNET	*EXTernal NETwork*
MIB	*Management Information Base*
JSON	*JavaScript Object Notation*
GNS3	*Graphical Network Simulator-3*
AMQP	*Advanced Message Queuing Protocol*
STP	*Spanning Tree Protocol*
OSI	*Open Systems Interconnection*
MPLS	*Multi-Protocol Label Switching*
Telnet	*TELetype NETwork*

CAPÍTULO 1

INTRODUÇÃO

Hoje em dia, no mundo atual, diz-se que este se está a tornar numa "Aldeia Global", isto devido ao enorme crescimento nos últimos anos das plataformas de Information Technologies (IT). Este crescimento rápido e constante é tal, que mesmo nas tarefas mais básicas se começa a usar equipamentos com capacidades computacionais com ligação à rede. Com este crescimento, é natural que o desafio de fornecer sistemas computacionais e redes que suportem todo este mundo ligado também aumente. Novas ideias e aproximações estão a aparecer para mitigar este problema. Algumas destas aproximações usam avançadas técnicas de virtualização, como a virtualização dos servidores e das redes que os suportam. Muitas das vezes, para quem já possui uma infraestrutura já em funcionamento, para manter este grau cada vez maior de exigência, os custos para migrar o equipamento para suportar virtualização é bastante alto. Tendo esta limitação em mente, a necessidade de desenvolver novas formas para tirar melhor partido da infraestrutura atual torna-se justificável e, consequentemente, o desafio aumenta devido à complexidade latente de adaptar uma infraestrutura mais antiga para responder aos mais recentes requisitos.
Ultimamente, e muito ligada também à virtualização, a buzzword Cloud começa a tornar-se muito popular. Esta define um novo paradigma de organização e gestão dos serviços IT, onde é possível ter capacidade de processamento e armazenamento na Cloud, que não é mais que um datacenter situado algures no mundo. Os serviços presentes na Cloud podem ser ou não acessíveis a todos consoante o tipo de Cloud em uso. Com este novo paradigma, o investimento em infraestruturas que o suportam torna-se um importante passo a tomar para quem deseje acompanhar e tomar parte nesta nova era nas ITs. Porém, o custo para atingir esse objetivo, em alguns dos casos, é muito elevado e torna perentória a necessidade de tentar adaptar as infraestruturas atuais a esta nova realidade. Este trabalho propõe uma solução para adaptar essas infraestruturas ao nível da rede na organização do paradigma Cloud.

1.1 MOTIVAÇÃO

A proliferação do paradigma de *Cloud Computing* foi capaz de trazer a necessária agilidade e flexibilidade para fazer face ao enorme crescimento do número de serviços baseados em IT. Essa agilidade e flexibilidade, é visível na organização e gestão efetuada dos recursos computacionais que fazem parte deste paradigma. O conceito de *Cloud Computing* é constituído por dois grandes grupos,

a *Cloud* privada e a *Cloud* pública. Os dois conceitos diferem na forma como os clientes têm acesso à mesma. Na *Cloud* pública, qualquer utilizador que o pretenda pode ter acesso à mesma, mediante pagamento ou não de algum valor ao operador pelo uso dos serviços. A *Cloud* privada é construída por uma entidade com vista a ser usada para seu próprio proveito ou para alguém no exterior usufruir. Por outras palavras, essa entidade pode construir a infraestrutura para ser usada pelos seus colaboradores ou então para ser alugada a outras organizações. A gestão da infraestrutura pode ser efetuada pela própria organização ou por alguma organização externa, sendo normalmente efetuada por um software de gestão de *Cloud*. Este software, é normalmente construído para funcionar em infraestruturas nas quais, os seus dispositivos integrantes possuem interfaces homogéneas compatíveis para poderem ser geridas pelo próprio. Porém, as organizações possuem já uma infraestrutura em produção que não pretendem abandonar (devido a custos, entre outras razões) e se fosse possível, seria uma mais valia poder integrá-la numa possível nova infraestrutura baseada em *Cloud*. Contudo, a infraestrutura existente (*legacy*) possui vários entraves à sua integração com a infraestrutura *Cloud*, pois a maior parte dos dispositivos nela existentes possuem normalmente um grande nível de heterogeneidade. A principal motivação deste trabalho reside em tentar dar mitigar as limitações da heterogeneidade desses dispositivos, de forma a que estes possam ser integrados na infraestrutura *Cloud*. Pretende-se encontrar um meio em que o software de gestão da *Cloud* possa efetuar configurações ao nível desses dispositivos, de modo a que, se possa estender os serviços presentes na infraestrutura *Cloud* para a infraestrutura *legacy* já existente e vice-versa.

As vantagens desta integração podem ser facilmente encontradas, como por exemplo, colocar no mesmo domínio de colisão um conjunto de máquinas virtuais que estejam na *Cloud* e um conjunto de máquinas físicas que estejam situadas na infraestrutura *legacy*. Isto pode servir para que as máquinas virtuais possam ter acesso a hardware específico presente nas máquinas físicas para efetuar uma tarefa que dele necessite. Esta junção da flexibilidade presente numa organização de uma infraestrutura baseada em *Cloud*, que permite uma gestão mais eficiente dos recursos e de serviços a si associados, com a possibilidade de ter acesso também a recursos presentes na infraestrutura *legacy*, traz imensos benefícios tanto aos operadores como aos utilizadores da infraestrutura. Posto isto, pode-se concluir de que se trata de uma área de grande interesse que carece de mais investigação e procura de novas soluções. Contudo, este tipo de integração de sistemas *legacy* é relativamente complexo, devido ao possível enorme número de dispositivos com diferentes interfaces e características. Este facto pode comprometer a eficiência e a fiabilidade da solução a adotar. A solução deve integrar-se da melhor maneira possível com os diversos softwares de *Cloud* existentes, fornecendo interfaces genéricas que permitam uma integração em diversos cenários e que a configuração dos dispositivos seja feita da maneira mais eficiente e fiável possível. Como exemplo, pode-se pensar na rede existente na Universidade de Aveiro, onde o *use case* de poder ter um computador localizado numa determinada sala de aula na mesma rede que um conjunto de máquinas virtuais localizadas na *Cloud* interna, pode trazer benefícios no contexto de ensino em determinadas situações. Um dos exemplos será o caso de ter um exame que necessite de uma sala com computadores, em que este necessite de acesso a máquinas localizadas no *datacenter*. Os principais objetivos a atingir com este trabalho são descritos nos quatro pontos seguintes.

- **Integração com o software de *Cloud* OpenStack** - Usar o software de *Cloud* OpenStack e as suas potencialidades de gestão centralizada de *Clouds* para uso e teste da solução a desenvolver.
- **Configuração *on-demand*** - A solução deve garantir a aplicação das configurações nos dispositivos de forma automática e apenas quando é necessário.
- **A aplicação das configurações tem de ser não destrutiva** - O processo de aplicação das

configurações nos dispositivos tem de ser não destrutivo, isto é, a aplicação de novas configurações não pode de forma alguma alterar/apagar as configurações existentes no dispositivo.

- **Configurável** - A solução deve ser o mais configurável e ágil possível, de forma a ser adaptável da melhor forma nos mais diversos tipos de software de *Cloud* e respetivos *deployments*.

1.1.1 ESTRUTURA DO DOCUMENTO

Este documento está organizado em sete capítulos. Este capítulo aborda as motivações para este trabalho de dissertação, em que se pretende desenvolver uma forma de estender as redes virtuais de um *datacenter* baseado em *Cloud* para redes externas *legacy*. O objetivo é alcançado aplicando configurações nos dispositivos da rede externa através de interfaces suportadas. Como plataforma de testes, vai ser usado o *Cloud Management Software* (CMS) OpenStack. Os restantes capítulos estão organizados da seguinte forma:

- Capítulo 2 (Definições) - Contém as principais definições de *background* que suportam este trabalho.
- Capítulo 3 (Estado da Arte) - Expõe o trabalho já efetuado na área, ou seja, as várias soluções já existentes nesta área de pesquisa de integração de redes *legacy* em ambientes *Cloud*.
- Capítulo 4 (Arquitetura da Solução) - Este capítulo aborda as questões relativas aos requisitos, *use cases*, e decisões de *design* tomadas para cumprir os requisitos.
- Capítulo 5 (Implementação da Solução) - Neste capítulo é detalhada a estrutura das ferramentas usadas para desenvolvimento e teste da solução. É descrita também, a abordagem tomada para implementação da solução.
- Capítulo 6 (Avaliação da Solução) - Neste capítulo são discutidos os resultados dos testes efetuados à solução proposta e as respetivas conclusões.
- Capítulo 7 (Conclusão) - Este capítulo contém uma análise geral à solução desenvolvida, os resultados obtidos, as lacunas principais e algumas considerações para trabalho futuro.

CAPÍTULO 2

DEFINIÇÕES

O objetivo deste capítulo é expor algumas das definições mais importantes relacionadas com este trabalho. Está dividido em definições conceptuais, que contém definições sobre conceitos e paradigmas acerca deste trabalho, e definições estruturais que contém as definições base usadas no desenvolvimento da solução.

2.1 DEFINIÇÕES CONCEPTUAIS

2.1.1 CLOUD COMPUTING

A palavra "Cloud" que começou por ser mencionada no mundo das IT no final do século XX como metáfora simbólica para a internet através de uma ligação telefónica. Mais recentemente esta palavra começou a ser predominantemente usada numa nova era na computação chamada de *Cloud* ou mais precisamente *Cloud Computing* (Computação na Nuvem) tanto por utilizadores que trabalham na área como fora dela, isto devido à rápida adoção e uso da mesma. Existem várias definições que apareceram no advento deste paradigma que o pretendem definir, mas nestes últimos anos as empresas no ramo e os utilizadores tendem a adotar uma definição criada pelo *National Institute of Standards and Technology* (NIST) que define [1] *Cloud Computing* como um modelo que permite a um conjunto de recursos computacionais possam ser atribuídos a um aglomerado de utilizadores através da rede. O modelo define também que os utilizadores têm a capacidade de pedir mais recursos quando assim o necessitem, sem precisarem de saber onde é que os recursos estão fisicamente localizados, sem nenhuma intervenção dos administradores da *Cloud* para os alocar e sem saberem de que forma é que os mesmos lhe foram fornecidos.

Cloud Computing retira a necessidade às empresas da criação da sua própria infraestrutura IT, desta forma esta não precisam de se preocupar com os seus custos de manutenção, problemas com escalabilidade entre outros possíveis problemas associados à posse da mesma. Isto permite que este esforço financeiro seja aplicado por exemplo no seu negócio diretamente, ou seja, permite à empresa focar-se mais no seu ramo específico de negócio. O uso da *Cloud* externa possui também a vantagem de ter uma maior propensão de escalabilidade e logo uma maior flexibilidade. Isto é, permite que de acordo com o volume de negócios da empresa, as suas necessidades de recursos computacionais possam

ser satisfeitas de uma forma mais dinâmica, sem preocupações com gastos extra e sem problemas de falta de recursos. Isto porque a empresa tem a hipótese de usar mais ou menos recursos, consoante as suas necessidades atuais pagando de acordo essa premissa. Por exemplo, no caso de a empresa possuir uma infraestrutura IT, se o volume de negócios diminui os recursos vão continuar a consumir a mesma quantidade energia sem estarem a ser usados, neste caso está-se num caso de sobre-dimensionamento, se o volume aumenta até um determinado nível corre-se o risco de sub-dimensionamento. Este cenário não se verifica se a empresa usar uma infraestrutura baseada em *Cloud* de um operador externo. Isto devido a este estar preparado para estes cenários, podendo alocar no caso de sub-dimensionamento os recursos a outro cliente rentabilizado ao máximo a infraestrutura. Este paradigma pode ser adotado em variados cenários e pode disponibilizar vários tipos de serviços, faz também uso de alguns conceitos de *Service-Oriented Architecture* como por exemplo divisão das aplicações em serviços para poderem ser usados por um maior número de utilizadores e assim melhorar a eficiência, possibilitando a sua reutilização. O NIST também identifica as suas principais características [1], que são as seguintes:

- **Pedidos *On-demand*** - Os utilizadores têm a possibilidade de reservar recursos para as suas necessidades automaticamente, sem qualquer intervenção para obtenção dos mesmos por parte do administrador da *Cloud*.
- **Acesso à rede sempre disponível** - O serviços presentes na *Cloud* são disponibilizados através da rede e o seu acesso é possível através de mecanismos definidos para suportar um grande número de clientes dos mais variados tipos (ex.: clientes móveis, portáteis, *workstations*, etc.).
- ***Pool* de recursos** - Os recursos devem estar disponíveis para serem acedidos por múltiplos utilizadores na *Cloud*. Estes são atribuídos aos utilizadores de acordo com as suas necessidades num determinado momento. O utilizador não sabe a localização onde são alocados os recursos nem possui qualquer controlo sobre isso mesmo.
- **Promover a elasticidade** - Os recursos atribuídos aos utilizadores podem ser aumentados ou diminuídos consoante as suas necessidades e devem ser atribuídos ou retirados de uma forma rápida e automática. O utilizador tem a perceção que os recursos a ele disponibilizados são infinitos e que estes lhe podem ser atribuídos a qualquer altura.
- **Medições de utilização** - A infraestrutura de *Cloud* deve possuir métodos para analisar e monitorizar o corrente uso dos recursos. Esta deve ter capacidade de apresentar estes valores por tipo de recurso (ex.: processamento, memoria, armazenamento, rede, etc.). Deve também fazer relatórios sobre os dados recolhidos de uma forma transparente tanto para os utilizadores como para os administradores. Estes dados recolhidos podem também ser usados, em certos casos, para custear a utilização dos recursos pelos utilizadores.

O paradigma de *Cloud Computing* é também dividido por modelos de serviço. Esta subdivisão é também definida pelo NIST [1] e é a seguinte:

- ***Software as a Service* (SaaS)** - Permite ao utilizador ter a capacidade de usar aplicações disponibilizadas pelo fornecedor de serviços, que estão a ser executadas numa infraestrutura de *Cloud* pertencente ao mesmo. Estas aplicações estão disponíveis através da rede por um cliente web ou por uma aplicação específica. O utilizador não toma parte na gestão da forma como as aplicações foram implementadas, nem da forma de como a infraestrutura é gerida para suportar

as aplicações. Apenas pode ajustar algumas definições relacionadas com o funcionamento interno da aplicação para o seu caso específico.

- ***Platform as a Service* (PaaS)** - Este modelo de serviço fornece ao utilizador a capacidade de instalar na infraestrutura de *Cloud*, as suas aplicações criadas numa linguagem de programação que seja suportada por esta. Assim como no SaaS, o utilizador não tem a capacidade de decidir onde e como as suas aplicações são executadas na *Cloud*. Tem apenas a possibilidade de controlar o funcionamento da sua própria aplicação, e algumas pequenas definições na plataforma onde a mesma é executada.
- ***Infrastructure as a Service* (IaaS)** - Neste tipo de *Cloud* o utilizador tem a capacidade de gerir alguns dos recursos da *Cloud* que lhe foram atribuídos. Estes recursos podem ser usados para instalar um sistema operativo e aí colocar o seu software em execução. Neste caso o utilizador controla o seu software e o sistema operativo por ele instalado. O acesso por parte do utilizador a definições é limitado, apenas tem acesso a algumas configurações de rede (ex.: firewalls, routers, etc.) e continua a não possuir privilégios para gerir a infraestrutura base da *Cloud*.

À data os serviços disponíveis nos diferentes modelos de serviço de *Cloud* eram os seguintes: Em SaaS os principais são as Google Apps[1] (Gmail, Google Drive, etc.), Microsoft 365[2], Salesforce CRM[3] e o Google Analytics[4]. Em PaaS existe o Google App Engine[5], Microsoft Azure[6], Amazon AWS[7], Engine Yard[8], OrangeScape[9] e o Mendix[10]. Nas alternativas *Free and Open-Source Software* (FOSS) existem opções como o RedHat OpenShift Enterprise[11], Cloud Foundry[12], Stackato[13] e o Cloudify[14] são alguns dos mais importantes. Em IaaS há o Rackspace[15], Amazon AWS, Microsoft Azure, Google Compute Engine[16], VMware vCloud Air[17] e o IBM SoftLayer[18]. Finalmente existem também alguns softwares IaaS FOSS que possuem uma grande popularidade que são o OpenStack[19], OpenNebula[20], Eucalyptus[21] e o Apache CloudStack[22].

O conceito de *Cloud Computing* é também subdividido pelo NIST [1] em quatro modelos diferentes de operação, alguns dos quais já foram anteriormente mencionados na secção 1.1 deste documento:

[1] https://gsuite.google.com
[2] https://products.office.com
[3] http://www.salesforce.com/eu/crm/what-is-crm.jsp
[4] https://www.google.com/analytics
[5] https://cloud.google.com/appengine
[6] https://azure.microsoft.com
[7] https://aws.amazon.com
[8] https://www.engineyard.com/
[9] http://www.orangescape.com
[10] https://www.mendix.com
[11] https://www.openshift.com
[12] https://www.cloudfoundry.org
[13] http://www8.hp.com/pt/pt/cloud/stackato.html
[14] http://getcloudify.org
[15] https://www.rackspace.com
[16] https://cloud.google.com/compute/
[17] http://vcloud.vmware.com/
[18] http://www.softlayer.com
[19] https://www.openstack.org
[20] https://opennebula.org
[21] http://www8.hp.com/us/en/cloud/helion-eucalyptus.html
[22] https://cloudstack.apache.org

- ***Cloud* privada** - Este tipo de *Cloud* é construida para ser usada por apenas uma organização, e os serviços fornecidos estão apenas disponíveis aos utilizadores que possuem um elo de ligação para com a organização (funcionários, gestores, etc.).
- ***Cloud* comunitária** - A *Cloud* deste tipo é normalmente partilhada por um conjunto de organizações que possuem algo em comum entre elas.
- ***Cloud* publica** - Neste tipo de *Cloud* os serviços disponíveis estão à disposição de quem os queira usar, podendo eventualmente estarem sujeitos a tarifas e outro tipo de limitações.
- ***Cloud* híbrida** - Trata-se de uma combinação entre os modelos de *Cloud* pública e privada. Neste caso a *Cloud* pode ter serviços privados acessíveis apenas a um grupo restrito de utilizadores e outros serviços públicos que podem ser acedidos por quem assim o pretenda.

2.1.2 *Service-Oriented Architecture* (SOA)

O paradigma de *Cloud Computing*, como já foi mencionado na secção 2.1.1, é composto por um conjunto de modelos de serviço, em que cada um deles implementa uma possibilidade diferente de os utilizadores tirarem partido da *Cloud*. Este facto leva-nos a outro importante conceito que se dá pelo nome de *Service-Oriented Architecture* (SOA).
SOA é uma arquitetura de software independente de marcas, produtos ou tecnologias que tem como objetivo definir um conjunto de regras para se poder subdividir uma aplicação em forma de vários serviços, serviços estes que normalmente possuem uma interface acessível através da rede. Este tipo de arquitetura promove a identificação de funcionalidades específicas de uma aplicação, e pretende que seja criado um serviço para cada uma delas. Desta forma as funcionalidades da aplicação podem ser aproveitadas por outros serviços e aplicações, promovendo assim a reutilização de código e uma melhor eficiência na gestão dos recursos computacionais. Estes serviços podem ser disponibilizados em diferentes computadores que cooperam entre si para que no final a lógica da aplicação seja da mesma forma alcançada como se este estive a correr apenas numa máquina. Esta arquitetura promove a reutilização de modelos de dados e funcionalidades comuns a várias aplicações. As consequências do uso desta arquitetura são uma redução de uso de recursos computacionais (memoria, processamento e armazenamento), redução do custo energéticos e de desenvolvimento, redução dos riscos inerentes ao desenvolvimento de aplicações *standalone* [2].
Algumas das suas características são exploradas mais em detalhe a seguir [3]:

- **Os serviços devem ser independentes** - Os serviços devem funcionar de uma forma totalmente autónoma uns dos outros, isto é, não devem ser dependentes uns dos outros para funcionarem.
- **Os serviços devem ser abstratos** - Os serviços devem ter um contrato o mais simples possível, com limites apertados sobre que informação deve estar disponível para o exterior.
- **Os serviços deve ser reutilizáveis** - Os serviços devem ser o mais abstratos possível, para que estes possam reutilizados noutros cenários e aplicações.
- **Os serviços devem ser autónomos** - Os serviços devem ser os únicos a terem o controlo total dos seus dados e do seu próprio ambiente de execução. Este facto melhora a robustez e a integridade destes.

- **Os serviços não devem possuir estados** - Os serviços devem guardar o mínimo de informação possível acerca dos seu estado interno atual, isto faz com que a quantidade de recursos e processamento necessários pelo serviço em execução seja reduzido, aumentando assim o tempo de resposta aos clientes.
- **Os serviços devem estar visíveis** - Os serviços devem conter os metadados suficientes para ser facilmente descobertos, e disponibilizarem a informação necessária para os clientes acerca da sua interface de utilização.
- **Os serviços são orquestráveis** - Um serviço normalmente é parte de uma solução de um problema maior, em que estes são criados com vista a dividir o problema em problemas mais pequenos e de mais fácil resolução. Desta forma, os serviços podem cooperar uns com os outros de forma a resolver o problema principal, e ainda ter o beneficio de poder ser reutilizados como parte da solução de outros problemas distintos.

2.1.3 NETWORK VIRTUALIZATION (VIRTUALIZAÇÃO DE REDE)

Perante o conceito de *Cloud* já mencionado na secção 2.1.1, consegue-se verificar que este assenta em muito no melhor aproveitamento possível dos recursos disponíveis. Uma das formas de alcançar um melhor aproveitamento dos recursos disponíveis é recorrendo à virtualização destes, por forma a torná-los disponíveis para um conjunto mais alargado de utilizadores. No caso da rede, devido à sua inerente natureza, não é possível virtualizar da mesma forma que outros recursos como por exemplo processamento. No entanto, esta também sofreu uma grande evolução com o aparecimento de novas técnicas de isolamento de tráfego, às quais podemos chamar de técnicas de virtualização de rede. *Network Virtualization* (Virtualização de Rede) é um termo que se refere a possibilidade de configurar múltiplas redes isoladas numa única rede física, recorrendo para isso ao uso do hardware de rede em conjunto com software específico [4]. Este conceito permite que numa rede física se possa definir cenários com redes virtuais de uma forma dinâmica. Isto é, é possível criar topologias de rede específicas em que cada uma dessas topologias é completamente isolada das restantes. Isto facilita assim, a criação de cenários de rede avançados sem a necessidade de estar a alterar constantemente a topologia da rede física. Torna-se então possível mascarar toda a complexidade da rede física agilizando também as tarefas de gestão da mesma, podendo algumas delas até serem feitas de forma automatizada. A virtualização de rede veio também dar aos administradores de rede, a possibilidade de segmentar a rede da forma mais conveniente para as suas organizações, de forma a preencher da melhor forma possível as suas necessidades, aliviando-os de estar constantemente a alterar infraestrutura de rede física[5]. Devido a estas características, a virtualização de rede veio abrir novos horizontes em relação ao tipo serviços de rede disponibilizados aos utilizadores. Para certos operadores permite-lhes ainda a possibilidade de oferecer serviços *end-to-end* aos seus utilizadores numa rede mais alargada (ex.: *Wide Area Network* (WAN)), como por exemplo ligar dois locais distintos geograficamente através da internet.

A virtualização de rede é caracterizada por as múltiplas tecnologias que a suportam, as mais importantes são [4] *Virtual Local Area Network*s (VLANs), *Virtual Private Network*s (VPNs), redes programáveis e redes *overlay*.

A VLAN é considerada umas das mais antigas tecnologias de virtualização de rede, quando apareceu trouxe a possibilidade de ter múltiplas *Local Area Network*s (LANs) na mesma ligação física através da configuração no software dos dispositivos, esta possibilidade veio trazer um grande nível de flexibilidade

em termos de gestão da rede física. A VLAN trouxe também outras vantagem como, segurança, isolamento de dados e o seu custo relativamente baixo de instalação(*Capital Expenditure* (CAPEX)) e operação (*Operational Expenditure* (OPEX)). A diferenciação entre as várias VLAN é feita com recurso a VLAN IDs que é incluído como parâmetro na camada dois do modelo *Open Systems Interconnection* (OSI) [6]

Outra forma bem conhecida de virtualização de rede é a VPN, que tem a capacidade de por exemplo, ligar dois ou mais edifícios de uma mesma empresa situados em localizações geograficamente diferentes, em que estes podem estar ligados por uma rede privada ou pública (ex.: internet). Este tipo de rede virtual resolve o problema da necessidade de ligar duas ou mais localizações diferentes, que por vezes só têm uma rede pública como possível meio de ligação, podendo assim criar uma ligação isolada sobre a rede pública entre os *endpoints*.

As redes programáveis são redes definidas por uma interface programável por software, que podem coexistir no mesmo dispositivo (pode ser um computador em forma de dispositivos virtuais ou por exemplo um switch físico compatível com a respetiva interface) [7]. Cada rede é definida através de uma interface de software, em que os dispositivos virtuais e/ou físicos são configurados de forma a que a topologia da rede seja definida como pedido. Cada rede criada possui as suas funcionalidades apenas visíveis e acessíveis nessa mesma rede, e portanto, as redes criadas desta forma são isoladas umas das outras mesmo estando localizadas no mesmo dispositivo. Este tipo de redes, devido às suas características são muito utilizadas nos CMS. O último tipo de virtualização de rede que é bem conhecido são as redes *overlay*, que fazem uso de tecnologias de *tunneling* (por exemplo *Generic Routing Encapsulation* (GRE) (L3) e em *Virtual Extensible LAN* (VXLAN) (L2oL3)) para isolar o tráfego entre os seus endpoints. Por outras palavras trata-se de uma rede construida em cima de outra rede [8]. Este tipo de redes permite que o tráfego possa atravessar várias redes de um *endpoint* para outro, sem a preocupação acerca de como a rede faz o seu encaminhamento. Este tipo de redes podem servir, por exemplo, para isolar o tráfego entre vários utilizadores numa rede partilhada por todos no *datacenter*. A virtualização de rede possui também uma vertente ligada à virtualização das funções de rede *Network Functions Virtualization* (NFV). Esta vertente, consiste na capacidade de mover as funções de rede (*routing*, *firewalls*, etc.) implementadas diretamente no hardware de rede, para software que corre num computador de uso geral. Esta mudança permite uma gestão da rede mais eficiente, uma redução importante na dependência dos vendedores de hardware, ao mesmo tempo que promove uma resolução de possíveis problemas de uma forma mais direta e rápida, facilitando o desenvolvimento de cenários de rede mais avançados.

2.1.4 SOFTWARE DEFINED NETWORKING (REDES DEFINIDAS POR SOFTWARE)

O conceito de *Cloud*, que pretende disponibilizar os recursos como um serviço aos utilizadores de uma forma ágil, recorre sempre que possível ao uso de tecnologias permitem maximizar esta agilidade de disponibilização dos recursos. A rede beneficia de um conjunto de novos paradigmas que tentam este objetivo [9], um deles é *Software-Defined Networking* (SDN). SDN é uma nova forma de arquitetura de rede, que tem como objetivo fazer com que a gestão da infraestrutura de rede seja mais centralizada, com esta aproximação o objetivo é simplificar o desenho da rede e a sua operação [10]. Este objetivo é alcançado movendo toda a lógica de gestão da rede e os protocolos presentes nos dispositivos, para um único ponto central e acessível na rede (pode ser por exemplo um computador de uso geral). Desta

forma, a gestão da rede fica mais simples, pois toda a configuração é feita num único ponto. Os dispositivos de rede podem ser mais simples, devido a apenas receberem instruções do controlador central. Além disso, os dispositivos são independentes do vendedor, ou seja, a interface dos mesmos é uniforme entre eles. Como consequência as organizações e respetivos administradores, ao terem o beneficio de possuírem ao seu encargo dispositivos de rede com uma interface uniforme, e que são geridos por um controlador centralizado, torna o seu trabalho mais simples e eficaz ao mesmo tempo que o custo é reduzido.

A arquitetura SDN, como é explicado em [10] e visível na figura 2.1, é dividida em três camadas distintas. As aplicações fazem pedidos de alterações na rede através da API fornecida pelo controlador. A camada de controlo recebe os pedidos das aplicações, e de acordo com as regras descritas nos pedidos, faz toda a lógica de processamento para obter as configurações necessárias a aplicar nos dispositivos de rede através da interface de controlo.

Com este tipo de arquitetura, resolve-se o problema de gerir múltiplas configurações diferentes em múltiplos dispositivos, tendo apenas um ponto central e uniforme para aplicação das mesmas. Com isto, tem-se o beneficio de conseguir definir cenários de rede mais rapidamente, usar múltiplas aplicações para fazer a gestão da rede fazendo uso da API disponibilizada. Uma rede baseada em SDN torna-se assim, mais flexível, mais ágil e simples de gerir. É possível ainda ter mecanismos inteligentes de orquestração, a operar através da API que podem responder automaticamente a certos eventos que podem ocorrer na rede, tornando a sua gestão mais autónoma.

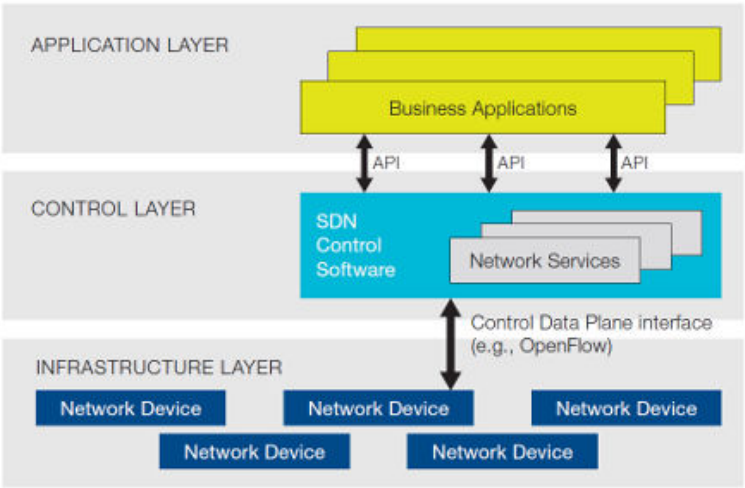

Figura 2.1: Software-Defined Network Architecture [10]

Alguns dos projetos mais importantes relacionados com SDN hoje em dia são:

- **OpenFlow** - O OpenFlow[23] é um protocolo de comunicação que é usado como interface entre o controlador e os dispositivos de rede [10]. A funcionalidade principal reside na possibilidade de manipular diretamente a camada de encaminhamento dos dispositivos de rede. Por outras palavras, tem como objetivo definir como é que o tráfego é encaminhado entre os vários dispositivos, impondo para isso regras com alto nível de granularidade.
- **OpenDaylight** - O OpenDaylight[24] é um software FOSS que tem por objetivo implementar um controlador SDN. Este é suportado pela *Linux Foundation* e principal seu objetivo é fornecer os meios necessários, para se construir uma base modular e flexível SDN [11]. É desenvolvido em Java, e como tal pode ser executado em sistemas operativos que o suportem.
- **Open vSwitch** - O Open vSwitch[25] é uma implementação *open-source* e multi-plataforma de um switch virtual, que possui na sua base o OpenFlow como principal protocolo para gestão de encaminhamento do tráfego.
- **Project Floodlight** - O Project Floodlight[26] é um controlador OVS FOSS que é suportado pela *Open Networking Foundation*. Tem uma grande comunidade a suportar o projeto com alguns colaboradores de grandes empresas, como por exemplo da *BigSwitch Networks*.
- **Ryu** - O Ryu[27] é uma *framework* SDN FOSS escrita em Python, que tem como objetivo facilitar o desenvolvimento de novas aplicações de gestão e controlo de rede.

2.1.5 FOG COMPUTING

Fog Computing é uma arquitetura que tem como objetivo colocar os serviços de rede o mais perto possível dos utilizadores, isto é, em vez dos serviços residirem apenas num ponto central da rede (por ex.: no *datacenter*), estes passam a poder estar mais próximos dos utilizadores finais, para por exemplo recolher dados sensoriais. Isto é possível fazendo com que parte do processamento e do armazenamento dos dados, seja feito nos dispositivos na rede mais próximos do utilizador. Estes dispositivos pode ser *set-top-boxes*, *routers*, *switches*, pontos de acesso, computadores portáteis, etc. [12]. Desta forma aumenta-se a redundância da informação, estando esta distribuída por diversos pontos da rede, promovendo assim de igual forma a colaboração entre dispositivos no processamento dos dados. Possui também a vantagem de, devido à menor distância entre o utilizador e o dispositivo onde se encontra a informação, o tempo de acesso aos dados seja menor em relação ao ter de aceder de uma forma constante ao *datacenter*. Possui igualmente vantagens ao nível do processamento, em que é reduzida a necessidade de enviar os dados para o *datacenter* para serem processados. Podendo parte do processamento destes dados ser efetuado nesses dispositivos, reduzindo assim também a carga da rede. Esta arquitetura é particularmente útil no contexto *Internet of Things* (IoT) [13] devido às vantagens mencionadas em relação ao processamento e armazenamento de dados. Estas vantagens são particularmente importantes em certas aplicações IoT, que exigem que os dados recolhidos de vários

[23]https://www.opennetworking.org/sdn-resources/openflow
[24]https://www.opendaylight.org/
[25]http://openvswitch.org/
[26]http://www.projectfloodlight.org/floodlight/
[27]https://osrg.github.io/ryu/

dispositivos, sejam tratados no menor intervalo de tempo possível. As principais características desta arquitetura são [14]:

- Os dados são gerados em dispositivos localizados em pontos próximos dos utilizadores. Este facto é particularmente importante em aplicações que requerem baixa latência.
- Promove um tipo de arquitetura descentralizada na qual, os dispositivos estão distribuídos por uma área geograficamente grande.
- Consegue acomodar um grande número de nós, consequência da distribuição geográfica dos nós. Facto este, mais evidente em redes sensoriais.
- Possui suporte à mobilidade, pois devido às características das aplicações baseadas em *fog computing*, estas recorrem muitas vezes à comunicação direta com dispositivos móveis.
- Interações em *real-time*, as mais importantes aplicações baseadas em *fog computing* interagem em *real-time*.
- O acesso é predominantemente *wireless*.
- É uma arquitetura heterogénea, isto é, é possível ter nós com diferentes características que podem ser aplicados nos mais variados tipos de circunstâncias.
- Interoperabilidade e federação. Os nós devem conseguir cooperar entre si de forma a garantir que determinados serviços/aplicações funcionem normalmente. Deve também garantir que os serviços/aplicações disponibilizados sejam federados no domínio em questão.
- Suporte para interoperabilidade com a *Cloud*. Como o *fog computing* tem como principal característica a habilidade de recolher e processar informação residente nas proximidades dos utilizadores, a interação com a *Cloud* torna-se uma mais valia para um posterior processamento e salvaguarda dessa informação.

2.1.6 LEGACY NETWORKS

Estas redes como o próprio nome indica, são redes legadas, isto é, são redes já existentes na infraestrutura que possuem na sua constituição dispositivos mais antigos. Estes normalmente, apenas suportam a aplicação de configurações de forma manual pelo administrador através das formas convencionais (por cabo no local, por telnet ou *Secure Shell* (SSH)). Estes dispositivos podem ser *switches* ou *routers* que possuem localmente as suas configurações e tabelas de encaminhamento. Quando é necessário fazer uma alteração na rede, o administrador da mesma necessita de aceder ao dispositivo a aplicar as configurações, para obter o comportamento desejado.

2.2 DEFINIÇÕES ESTRUTURAIS

2.2.1 *cloud management software* (SOFTWARE DE GESTÃO DE CLOUD)

O software de gestão de *Cloud* pode ser visto como sendo o coração de uma infraestrutura IT baseada em *Cloud*. Tem a responsabilidade de gerir um conjunto de recursos computacionais à sua

disposição (processamento, rede, armazenamento, etc.). Em que esta gestão deve ser feita de uma forma o mais eficiente possível, de modo tirar o máximo aproveitamento possível dos recursos por ele geridos. Hoje em dia essa gestão é feita de uma forma cada vez mais automática, isto também, é resultado do aumento das necessidades de serviços baseados em IT. Este facto conduz a um aumento da pesquisa por novas formas mais eficientes e automatizadas na gestão dos recursos, por forma a manter os custos de operação (OPEX) baixos [15] e ao mesmo tempo aumentar a disponibilidade dos serviços.

Por norma esses recursos são disponibilizados por utilizador, e é da responsabilidade do software a gestão do *life-cycle* dos recursos que são atribuídos a cada um deles. Por exemplo numa IaaS, quando um utilizador cria uma nova *Virtual Machine* (VM), é da responsabilidade do software de gestão de *Cloud* a reserva dos recursos [16] para essa VM. Se por acaso o utilizador posteriormente necessitar de mais recursos para essa VM, o software tem de verificar se é possível obtê-los e, caso seja possível, atribui-los. Quando a VM deixar de ser necessária os recursos devem ser libertados e colocados à disposição dos restantes utilizadores. Tudo isto deve ser efetuado de forma rápida, dinâmica e totalmente transparente para os utilizadores.

Normalmente, e por uma questão de eficiência, organização e garantia de alta disponibilidade, este software é subdividido num sub-conjunto de módulos de software em que cada um é responsável por um tipo de recurso específico. Por exemplo, podem existir módulos para gerir a rede, outros para gerir o processamento e outros para gerir o armazenamento.

De uma forma mais detalhada, são agora enunciadas algumas das mais importantes características que um software de gestão de *Cloud* deve possuir [16]:

- **Suporte para virtualização** - O software deve possuir capacidades para gerir todo *Life-cycle* relacionado com as VMs. Deve reservar e gerir todos os recursos necessários ao seu funcionamento, (*Central Processing Unit* (CPU), memória, rede, armazenamento, etc.) e deve ter uma interface uniforme para controlo de um conjunto alargado de *hypervisors*.
- **Os recursos devem poder ser reservados *On-demand*** - O software deve fornecer formas eficazes de reservar e gerir os recursos, sem a necessidade de intervenção por parte dos administradores da infraestrutura. Entenda-se por intervenção, a necessidade de aplicação de configurações por parte dos administradores, para colocar um recurso disponível a um utilizador.
- **Virtualização de acesso ao armazenamento** - O software deve ser capaz de criar uma camada de abstração lógica dos dispositivos de armazenamento, de forma a não dar acesso direto ao suporte físico. Isto permite a possibilidade de serem criados discos virtuais que são totalmente independentes da localização física do armazenamento. Normalmente, estes discos virtuais são disponibilizados por uma *Storage Area Network* (SAN) através de protocolos como o *Fiber Channel*, iSCSI e *Network File System* (NFS).
- **Virtualização de redes** - Como foi mencionado na secção 2.1.3, esta consiste na criação de redes isoladas em cima da rede física, de forma a tornar possível isolar o tráfego dos utilizadores. O software deve ter a capacidade de configurar os dispositivos de rede existentes (tanto baseados em hardware como em software), para se criar topologias de rede totalmente isoladas umas das outras.

OPENSTACK

O OpenStack[28] é um dos softwares de gestão de *Cloud* FOSS mais usados a nível mundial, que tem como foco primário a gestão de *Clouds* do tipo IaaS. Este projeto foi idealizado numa primeira fase através de uma iniciativa de lançar uma plataforma *Cloud* open-source, esta iniciativa foi tomada por duas empresas a NASA[29] e a Rackspace[30] hosting. Em 2010 lançaram em conjunto com a comunidade, aquilo que seria a primeira versão deste projeto conjunto, que permite usar hardware standard para disponibilizar serviços *Cloud*. A primeira versão tinha o nome de código *Austin* e tinha updates num ciclo mensal. As primeiras porções de código tiveram a proveniência de projetos das duas empresas fundadoras, o NASA *Nebula* e o *Rackspace Cloud Files Platform*.
O projeto está escrito em Python[31], pode correr em sistemas Linux e hoje em dia conta com *releases* estáveis de 6 em 6 meses. Aquando da escrita deste documento o projeto estava na versão com o nome de código *Mitaka*. O projeto é gerido por uma hierarquia constituída pelos chamados membros *Platinum*, *Gold* e membros individuais. Sendo os membros *Platinum* os com mais influência na definição do percurso do projeto e os membros individuais com menos. Como membros *Platinum* o projeto tem colaboradores de empresas como a SUSE, Hewlett Packard (HP), Red Hat, Intel, IBM, entre outros. Na realidade o OpenStack não é um software por si só, mas sim um conjunto de componentes/serviços em que cada um tem uma funcionalidade bem definida. Os principais são:

- **Nova** - É o componente responsável pela gestão dos recursos de processamento da plataforma (CPU e RAM) e um dos mais importantes. Por outras palavras, é responsável pelo *life-cycle* das VMs dos utilizadores. O Nova pode orquestrar tecnologias de virtualização como o *Kernel-based Virtual Machine* (KVM), soluções VMware, Xen, Hyper-V e LXC.
- **Keystone** - O Keystone tem como objetivo disponibilizar um meio centralizado de gestão de contas, autenticação e permissões de acesso aos serviços. Os clientes, administradores e restantes componentes da *Cloud* OpenStack acedem a este por meio de uma API RESTful. O Keystone por si só não fornece meios de gestão de utilizadores, em vez disso delega essas funções a serviços já existentes que o façam. Tem compatibilidade com diversos métodos de autenticação tais como username e password (por exemplo serviços *Lightweight Directory Access Protocol* (LDAP), se já existentes na infraestrutura), autenticação por token, autenticação *multi-factor* e através do Kerberos.
- **Neutron** - Este é o componente responsável pela gestão das redes virtuais dos utilizadores da *Cloud* gerida pelo OpenStack, tem como objetivo fornecer uma solução *Network as a Service* (NaaS) no contexto da plataforma OpenStack. Permite aos utilizadores da *Cloud* criarem redes IP com topologias complexas para interligar as suas VMs, isto de uma forma totalmente isolada de cada um dos utilizadores e sem problemas com limites de portas e/ou dispositivos. Permite ainda a criação de routers virtuais para acesso das redes dos utilizadores ao exterior, com recuso a *Network Address Translation* (NAT) ou através de mapeamento para IPs disponibilizados pelo operador da infraestrutura (*Floating IP*). O Neutron recorre a tecnologias de virtualização de rede como VLANs, GRE e VXLAN para garantir o isolamento entre as redes. Fornece ainda serviços de gestão de endereços IP como o *Dynamic Host Configuration Protocol* (DHCP), e possui capacidades de integração com plataformas SDN como o OpenFlow,

[28]http://www.openstack.org/
[29]http://www.nasa.gov/
[30]https://www.rackspace.com
[31]https://www.python.org

que desta forma, lhe proporciona altos níveis de escalabilidade. Na secção seguinte a estrutura interna do Neutron vai ser analisada mais ao pormenor, pois é o componente onde a *framework* vai ser integrada.

- **Glance** - O Glance é responsável pela gestão das imagens de instalação das VMs, isto é, permite o registo de novas imagens e a disponibilização das mesmas ao Nova para criação de novas VMs. As imagens podem ser vistas como templates para criação de novas VMs. Permite igualmente a gestão de imagens de backup em número ilimitado. Este componente fornece uma API RESTful que permite, fazer pesquisa de imagens e fazer a transferência de imagens aquando da criação de novas VMs.
- **Cinder** - O Cinder disponibiliza um serviço de *block-storage*, ou seja, é responsável pela gestão dos dispositivos de armazenamento por blocos que são usados nas VM, como discos rígidos para armazenamento de informação. Esta gestão é compreendida pelas funções de criação de volumes, associação e dissociação destes perante as VMs. Esta gestão está perfeitamente integrada com o Nova e com a sua API RESTful, permitindo aos utilizadores fazer uma gestão conveniente do espaço disponível, gerindo os seus volumes da forma que mais lhe convém. Para além do armazenamento local da máquina onde está instalado o componente, possui ainda meios para integração com infraestruturas de armazenamento já existentes e concebidas para o efeito.
- **Horizon** - O componente Horizon tem como objetivo disponibilizar uma interface web simples e intuitiva, onde seja possível configurar e gerir todos aspetos relacionados com as funcionalidades do OpenStack. Permite que administradores e utilizadores tenham acesso a uma interface gráfica simples e fácil de usar. Esta interface foi construida de forma a ser expansível e configurável, por forma a atender a requisitos específicos por parte da organização que gere a *Cloud*. Estes requisitos podem ser funcionalidades como por exemplo, mecanismos de cobrança, contadores de uso de recursos, entre outros.
- **Swift** - O Swift tem como tarefa guardar e disponibilizar dados e objetos não estruturados através de uma API RESTful. O Swift tem a capacidade de guardar os dados de uma forma distribuída por vários servidores, proporcionando uma elevada redundância dos dados.

Uma grande parte destes componentes apresentados anteriormente possui uma API RESTful própria, que é usada como forma de interação com outros componentes, à exceção do Horizon que é cliente das APIs dos outros componentes por ser um *dashboard*. Os componentes delegam as operações de autenticação e verificação de permissões de acesso às funcionalidades no componente Keystone.
Estes componentes possuem características totalmente modulares, o que permite uma instalação com grandes graus de flexibilidade e alta disponibilidade onde o administrador da *Cloud* pode escolher quais os que pretende instalar, consoante os objetivos da *Cloud* a implementar. No entanto, para se conseguir ter uma *Cloud* funcional, alguns destes componentes são um requisito mínimo. Normalmente estes componentes estão distribuídos por nós de computação que são máquinas x86 com o *Operating System* (OS) Linux. Esta divisão por nós, permite que os componentes tenham uma melhor performance e um melhor isolamento entre eles, ao residirem em sistemas diferentes. Na última versão (Mitaka), o número de nós e componentes mínimos que uma instalação OpenStack deverá possuir são os seguintes:

- **Controller Node** - É o nó que concentra as funcionalidades de gestão mais criticas da *Cloud*, em que componentes instalados são os descritos a seguir. O *dashboard* Horizon que fornece uma interface gráfica para gestão da *Cloud*. O *Neutron* que faz toda a gestão das redes dos

utilizadores. O *Keystone* que gere os mecanismos de autenticação, autorização e contas dos utilizadores. E por fim o *Glance* que faz a gestão e fornecimento das imagens de disco para as VMs. Estes são os componentes base que o *Controller Node* tem de incluir. Podem no entanto serem instalados outros para proporcionar funcionalidades específicas no contexto de cada *Cloud*. Por exemplo o *Cinder* (*block storage*) e *Swift* (*object storage*) podem ser instalados no *Controller Node*, ou serem colocados em máquinas diferentes. Isto vai depender, como já foi mencionado, dos níveis de disponibilidade a atingir, do hardware disponível, entre outras razões. Por exemplo o *Cinder* (*block storage*) e *Swift* (*object storage*) podem ser instalados no *Controller Node*, ou serem colocados em máquinas diferentes. Isto vai depender como já foi mencionado, dos níveis de disponibilidade a atingir, do hardware disponível, entre outras razões.

- **Network Node** - Este nó alberga o servidor de DHCP e os routers virtuais. Estas funcionalidades são geridas por agentes do Neutron a correr nesse nó. Este nó na versão Mitaka é facultativo, as suas funcionalidades podem ser integradas no *Controller Node*. Apenas se considera o seu uso em *deployments* onde é necessário garantir alta disponibilidade.
- **Compute Node** - Este nó contém como componente o *Nova* para criação e gestão do *life-cycle* das VMs. Podem existir um ou vários *Compute nodes* por instalação consoante as necessidades. Estes nós são dos mais importantes na infraestrutura *Cloud* por serem onde as VMs são colocadas em execução.

Um exemplo desta instalação minimalista pode ser visualizada na figura 2.2.

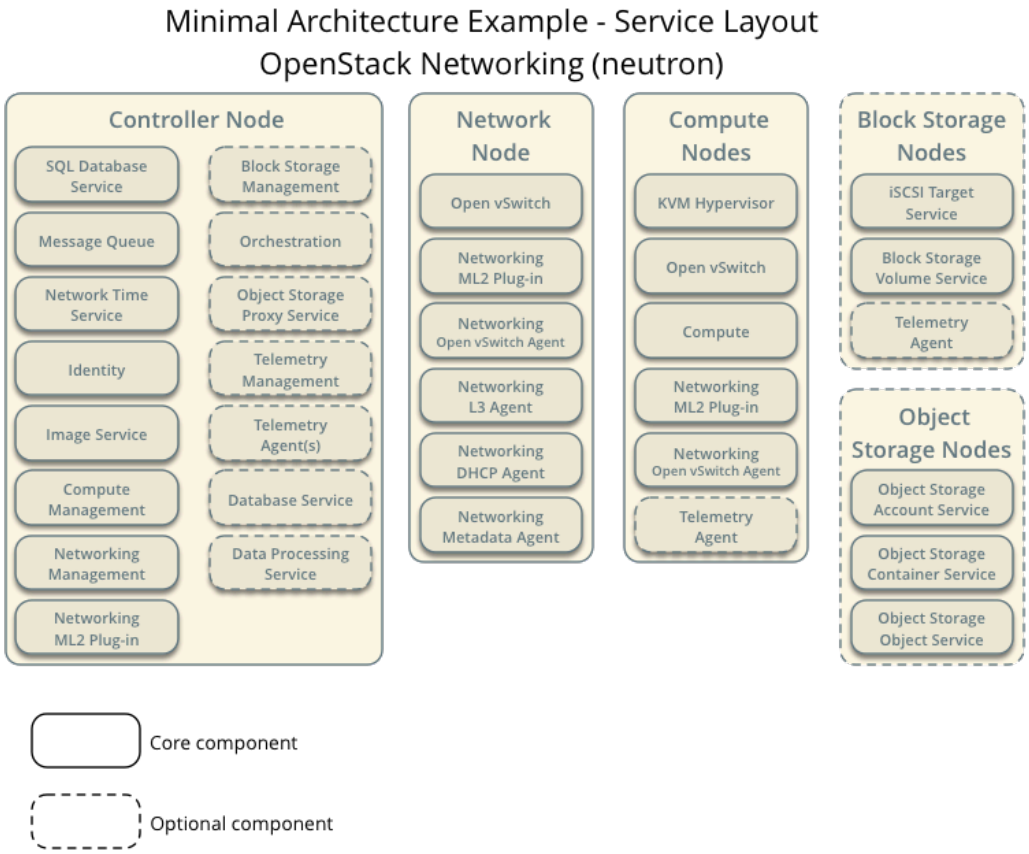

Figura 2.2: Exemplo de uma instalação minimalista do OpenStack com o core plugin ML2 [17]

O projeto OpenStack conta ainda com um projeto chamado *DevStack*, que tem como objetivo proporcionar uma forma de obter uma instalação OpenStack para desenvolvimento numa única máquina (ou até mesmo em várias usando os ficheiros de configuração e vários DevStack em várias máquinas), de uma forma rápida e simples. O DevStack é configurável através da edição de ficheiros de configuração, e a sua execução é feita através de um script Shell que faz todo o processo necessário na máquina pretendida. O DevStack possui a capacidade de escolher quais dos serviços a serem instalados na máquina podendo assim, ser usado para fazer *deploy* de um conjunto de serviços numa máquina, e outros noutra. É assim então possível criar um ambiente de testes com vários nós, cada um com um conjunto de serviços previamente definidos.
Para teste da *framework* proposta neste trabalho, foi escolhida a última versão do OpenStack a versão *Mitaka*, com o DevStack como ferramenta de criação do ambiente para testes de funcionamento. Importa ainda referir, que o projeto OpenStack define *guidelines* [18] para o desenvolvimento no projeto. Neste trabalho, houve um esforço para que durante o desenvolvimento essas *guidelines* fossem cumpridas.

capítulo 3

Estado da Arte

Neste capítulo é exposto o estado da arte de integração de redes legacy com as frameworks de Cloud Computing. São analisadas algumas das soluções já existentes em relação às suas vantagens e desvantagens. Com esta análise pretende-se obter algum background acerca desta área em particular, e obter algumas linhas de orientação para este trabalho ser bem sucedido.

3.1 colocar servidores físicos em redes virtuais

Em 2013, Bruce Davie desenvolveu uma solução [19] que permitia colocar servidores físicos (que não podiam ser movidos para máquinas virtuais devido a razões como por ex.: aplicações que necessitavam de ter o menor tempo de execução possível, aplicações que necessitavam de acesso direto a hardware específico, etc.), no mesmo domínio de colisão de redes virtuais geridas pelo VMware NSX[1]. Como primeira hipótese usou uma máquina x86 para servir de OVS *gateway*; este *gateway* era controlado pelo NSX fornecendo-lhe as necessárias configurações, de forma a este ser capaz de mapear as redes virtuais geridas pelo NSX com VLANs, que eram configuradas em portas de acesso físicas. A sua equipa conseguiu este objetivo, usando o protocolo Open vSwitch Database (OVSDB) como transporte das configurações para o OVS *gateway*. Estas configurações visavam a criação de túneis VXLAN, para transporte do tráfego de forma isolada entre os *hypervisors* e o OVS *gateway*. Esta solução é tecnicamente válida mas, se a quantidade de tráfego e o número de túneis aumenta em determinado nível, é fácil chegar ao limite de carga da máquina que está a fazer de *gateway*. Isto porque, a máquina não consegue processar todo tráfego que flui pelas portas do switch virtual; assim a equipa concluiu que esta solução não era escalável e consequentemente não viável.
No entanto, por volta da mesma altura, novos dispositivos de rede começaram a surgir com suporte integrado para criação de *endpoints* de túneis VXLAN, a estes dispositivos foi dado o nome de *VXLAN Tunnel End Point*s (VTEPs) devido a essa capacidade. Tendo hipótese de ter acesso direto ao mapeamento entre *endpoints* virtuais e as portas físicas no hardware, resolve-se o problema que existia com o uso da máquina x86 a fazer de OVS *gateway*. Mas à custa disso, o número de mensagens adicionais enviadas para fazer o *setup* dos VTEPs aumenta, isto devido à necessidade de informar o controlador NSX, acerca da localização dos endereços *Media Access Control* (MAC) ligados a cada

[1]http://www.vmware.com/products/nsx.html

porta física do switch. Esta arquitetura está representada na figura 3.1.
Estas duas aproximações têm vantagens e desvantagens, fazendo uso da máquina x86 a solução é mais *future proof*. Isto devido à facilidade com que novas funcionalidades que possam vir a existir no futuro, sejam possíveis de ser integradas mais facilmente por software. Mas contudo, os problemas de escalabilidade são evidentes, se o volume de tráfego ou o número de túneis/portas aumenta.
Na solução baseada em VTEPs, existe a possibilidade de ficar de alguma forma dependentes do vendedor do hardware com suporte VTEP. Estas duas soluções apresentadas são direcionadas para redes *Layer-2* (L2); um ano mais tarde Bruce Davie agora com Ken Duda desenvolveram uma solução direcionada para redes *Layer-3* (L3) [20].

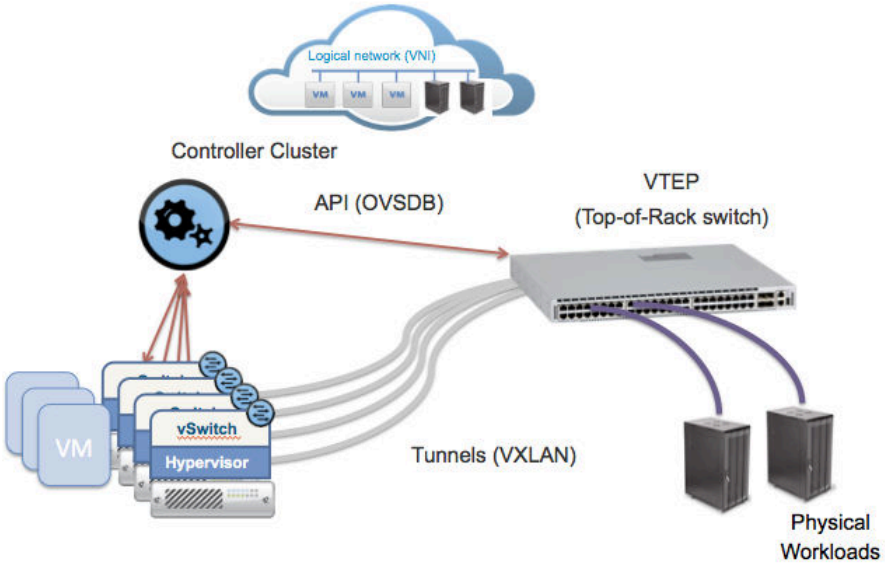

Figura 3.1: Arquitetura do hardware VTEP

Ambas as soluções resolvem de facto o problema de colocar os servidores físicos em domínios de colisão virtuais. Mas possuem alguns problemas que impedem o seu uso num contexto mais alargado, como por exemplo, a dependência do uso do VMware NSX e do protocolo *ovsdb*. Por outras palavras, esta solução não é modular ao ponto de não ser extensível ao uso de outras tecnologias e protocolos. Assim sendo, são soluções difíceis de exportar para outros tipos de ambientes de *Cloud*.

3.2 LEGACYFLOW - UMA FORMA DE GERIR A REDE *legacy* COM O *openflow*

Em 2012, um ano depois do aparecimento da primeira versão do *OpenFlow*, que tem um papel importante nas redes de próxima geração, Fernando Farias propôs uma solução [21] que pretendia adaptar os dispositivos de rede *legacy* ao paradigma SDN, que as empresas por exemplo, devido a custos de migração para equipamentos mais recentes, pretendem manter em funcionamento. Como mencionado na secção 2.1.6, é possível definir dispositivos *legacy* como dispositivos cuja configuração,

é feita de forma tradicional pelo administrador usando interfaces CLI, SSH ou telnet. Neste caso, os autores desenvolveram uma solução que transforma os dispositivos *legacy*, em dispositivos compatíveis com o protocolo *OpenFlow*. Assim, é dada a possibilidade de ter uma plataforma de gestão de rede uniforme, baseada num controlador que suporte *OpenFlow*. Esta solução resolve muitos problemas relacionados com a migração da infraestrutura de rede, permitindo assim, que se possa ter um processo faseado de substituição dos dispositivos *legacy* por outros compatíveis com *OpenFlow* de acordo com possibilidades.

À solução foi dado o nome de *LegacyFlow* e está dividida em duas camadas distintas. Cada camada possui um processo diferente para tornar a solução mais modular e para melhor lidar com diferentes tipos de dispositivos *legacy*. Estas duas camadas comunicam entre si através de *Inter-Process Communication* (IPC). Uma das camadas é chamada de *Legacy datapath*, que consiste num novo *datapath* desenvolvido especificamente para lidar com os dispositivos *legacy*. O *datapath*, tem como objetivo receber instruções do controlador *OpenFlow*, e aplicar as configurações necessárias nos dispositivos *legacy* através do *switch controller* que corresponde à segunda camada. Sobre o *switch controller*, como o próprio nome sugere, trata-se de uma máquina responsável pela aplicação das configurações nos dispositivos *legacy* através de uma interface compatível (ex.: SSH, telnet ou *web services*). O *switch controller* tem também a responsabilidade de fornecer informações necessárias ao controlador *OpenFlow*, acerca das interfaces destes dispositivos para este poder criar interfaces virtuais que as representem no seu contexto interno. Nos testes efetuados, esta recolha de informações das interfaces é feita pelo *switch controller*, recorrendo ao protocolo *Simple Network Management Protocol* (SNMP). Os circuitos criados nos dispositivos *legacy* são construidos usando VLANs.

A solução fornece de facto, uma forma de integrar os dispositivos *legacy* numa rede baseada em *OpenFlow*. Proporciona a possibilidade de desenvolver *switch controllers* compatíveis com os mais variados tipos de dispositivos. No entanto, esta solução só é viável em redes baseadas em *OpenFlow*, portanto tal como a solução mencionada na secção anterior, a integração com outros tipos de ambientes de rede não é possível, ou é de difícil execução.

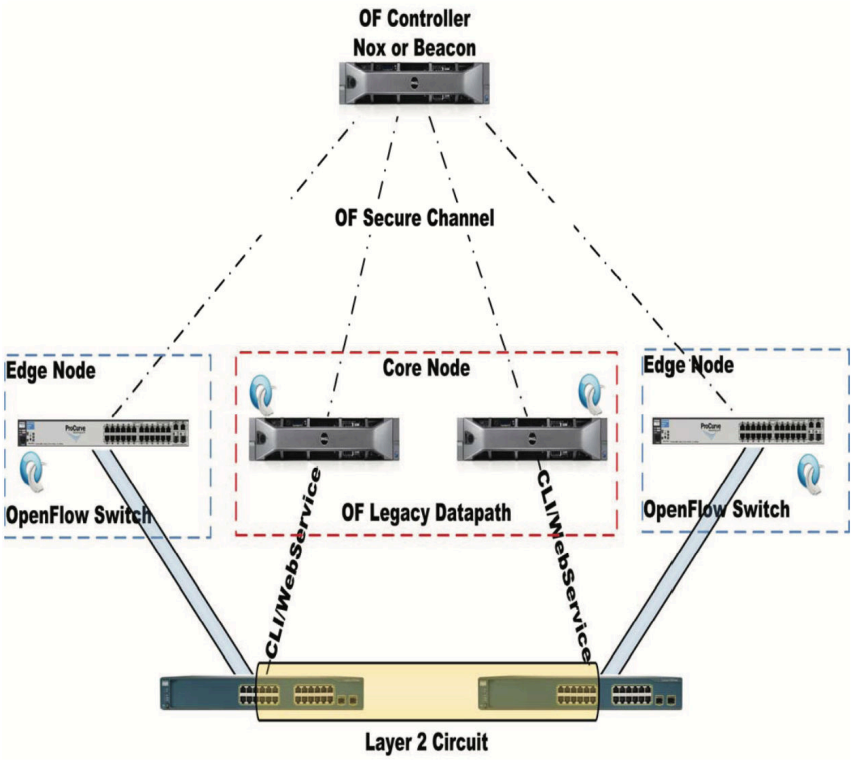

Figura 3.2: LegacyFlow - uma forma de adaptar os dispositivos *legacy* para serem usados em redes baseadas em *OpenFlow*.

3.3 CONTROLO DE INFRAESTRUTURA DE REDE PARA CAMPUS VIRTUAIS

No ano de 2013, Filipe Manco na sua dissertação de mestrado [22], propôs uma arquitetura para uma *framework* que iria permitir estender as redes virtuais, geridas por um CMS, para a rede física constituída por dispositivos *legacy*. O seu principal objetivo era configurar esses dispositivos, de modo a criar rotas isoladas de uma ou várias redes virtuais, geridas pelo CMS. Estas extensões, teriam como *endpoint* qualquer sitio da rede, onde houvesse dispositivos que necessitassem de ter acesso L2 às redes virtuais. Estas rotas seriam criadas com recurso a tecnologias de virtualização de rede, tais como VLAN, GRE, VXLAN, ou outras que tenham a capacidade de isolar tráfego L2 numa rede física, e que fossem suportadas pelo dispositivos. A solução previa também que estas extensões fossem aplicadas de forma automática e transparente para o utilizador. Estando a *framework* responsável pela descoberta dos dispositivos existentes na rede física, e posterior escolha dos segmentos para construir o caminho para as extensões a serem criadas, esta escolha do caminho e a descoberta dos dispositivos era controlada por quem administra o sistema. O administrador tem assim a responsabilidade de escolher os algoritmos e configurações pretendidas, de forma a atingir o comportamento desejado pela descoberta dos dispositivos e escolha dos segmentos relativos às extensões. A solução é composta por um misto de uma solução *standalone* e por recurso a extensões à API do CMS. Desta forma seria possível obter o melhor dos dois mundos. Assim, tendo a flexibilidade de uma solução *standalone* consegue-se ter uma

framework mais abrangente, ao permitir outros elementos gerir o seu comportamento sem ser somente o CMS. Ao ser também uma extensão ao CMS facilita a sua integração, e com isto, é possível ter acesso aos recursos já existentes como base de dados e API. Devido a este último facto, consegue-se fazer uso da informação sobre utilizadores, redes virtuais, cotas de utilização entre outras. Com esta arquitetura mista consegue-se ter o CMS a gerir os pedidos de criação das extensões (por exemplo pedidos de portas na rede física) e outro agente externo a fazer a gestão e monitorização dos dispositivos *legacy*. A arquitetura apresentada consegue na realidade, representar a rede física usando entidades bem definidas. Possui ainda, a capacidade de ter a informação necessária para se poder criar e gerir ligações virtuais nos os dispositivos da rede física, com a devida associação às redes virtuais já existentes na *Cloud*. O possível problema que esta solução possa ter, prende-se com o facto de poder ser pouco escalável devido à quantidade de informação que é preciso gerir relacionada com a rede física. Com uma topologia de rede de uma determinada dimensão, a gestão desta informação pode-se tornar bastante complexa. No entanto, como esta arquitetura não chegou a ser implementada e testada, não existem resultados que sustentem estas afirmações.
Esta dissertação tem por base esta arquitetura idealizada pelo Filipe e segue, de certa forma, os trâmites contidos na *Blueprint* por ele proposta [23]. Nos próximos capítulos são discutidas mais a fundo as vantagens e as desvantagens da mesma.

3.4 ESTENDENDO AS REDES VIRTUAIS OPENSTACK NEUTRON USANDO PORTAS EXTERNAS

Em 2014, Igor Cardoso na sua dissertação de mestrado [24], desenvolveu uma solução que permite estender as redes virtuais para a rede física não gerida pelo CMS. A solução proposta é baseada na criação de *External Ports*. A arquitetura da solução é composta por mais duas entidades, os *Attachment Devices* e os *Attachment Points* que suportam a criação das *External Ports*. A solução consiste na criação de túneis *overlay* (GRE ou VXLAN), entre a infraestrutura *Cloud* e os chamados *Attachment Devices*. Os *endpoints* dos túneis do lado da rede física externa dão acesso a *Attachment Points*. Estes *Attachment Points* que são disponibilizados pelos *Attachment Devices*, tratam-se de entidades lógicas do ponto de vista do CMS. Estes podem representar uma única porta física, um conjunto de portas físicas que vão dar acesso à mesma rede virtual (por exemplo usando VLANs). Uma porta lógica (por ex.: uma porta de um switch virtual) ou um conjunto de portas lógicas idênticas com características idênticas ao conjunto de portas físicas apresentado anteriormente. Ou até ainda um *Service Set Identifier* (SSID) de uma rede Wi-Fi. O *Attachment Device* por sua vez é um dispositivo com capacidade de ser configurado remotamente e com suporte das tecnologias *overlay* GRE ou VXLAN, este suporte é requisito fundamental para se poder criar *Attachment Points*. Um exemplo de um possível uso desta solução, está representado na figura 3.3.
A solução desenvolvida permite a extensão de redes virtuais presentes na *Cloud* para fora deste domínio, dando a possibilidade de dar acesso L2 a dispositivos situados numa rede externa a uma qualquer rede virtual existente na *Cloud*. No entanto, a solução será mais eficaz para o caso de haver a necessidade de, estender uma rede virtual para uma localização em que a ligação necessite de passar por WANs (por ex.: a Internet). Por outras palavras, a solução é mais indicada para o caso de haver a necessidade de ligar duas localizações que necessitem de passar por múltiplas redes heterogéneas. Por exemplo, uma *Telecommunications Company* (TELCO) que pretende ter um maior controlo nos seus dispositivos

que estão instalados nas habitações dos seus clientes, e que, desta forma consegue aceder aos seus dispositivos para aplicação de configurações. Assim estas empresas, podem tirar todo o partido de o poder fazer de uma forma flexível e integrada através de uma infraestrutura *Cloud*. Por outro lado, para estender redes virtuais para redes de menor dimensão como redes campus, este solução pode não ser a ideal, devido ao maior número de dispositivos e consequente possível maior número de *External Ports* a poderem ter de ser criadas. Como consequência, e por ser uma solução baseada em túneis *overlay*, exige dos dispositivos maior capacidade de processamento. Outro problema subjacente é o facto de estar, de alguma forma, limitado somente ao uso de tecnologias de *tunneling* o que restringe o seu uso a dispositivos que as suportem.

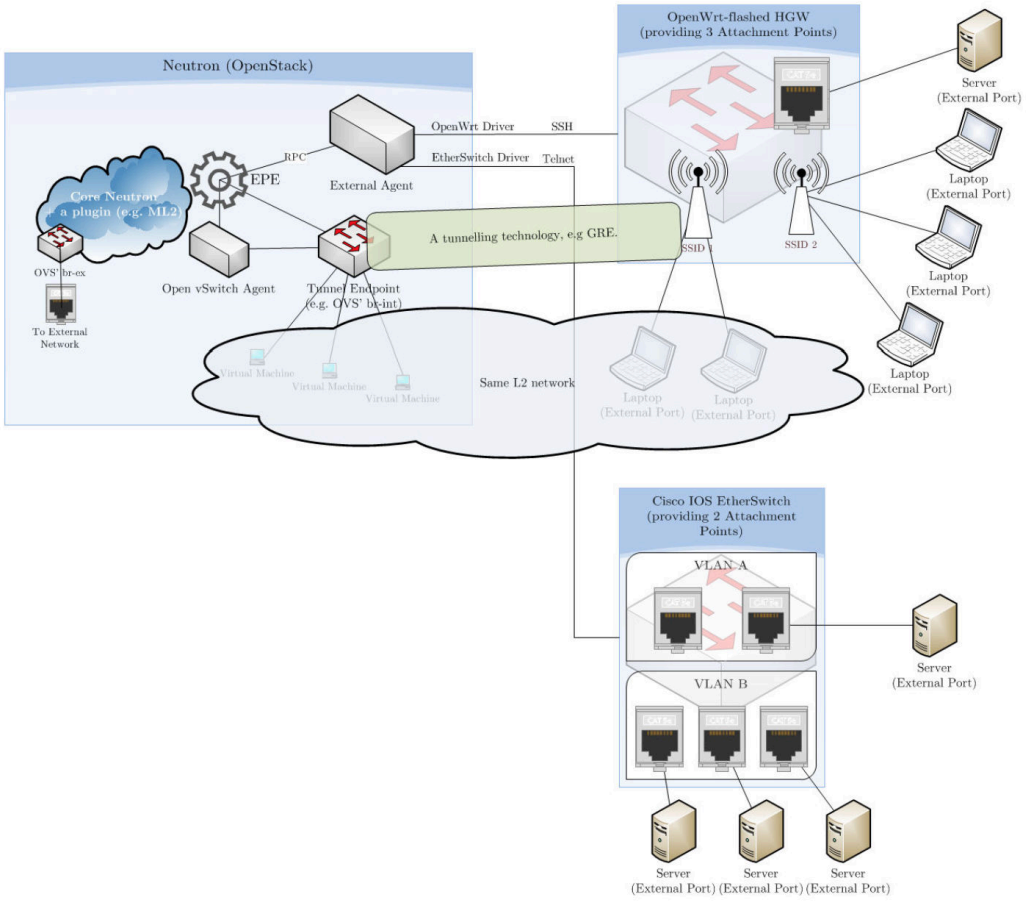

Figura 3.3: Exemplo do uso da extensão External Port [24]

3.5 OPENSTACK NEUTRON - LAYER-2 GATEWAY SERVICE PLUGIN

L2-GW é um sub projeto ligado ao plugin ML2 do OpenStack Neutron que tem como objetivo, tal como a solução anterior, unir dois domínios L2 de forma a obter um único domínio de colisão. Em que um deles representa uma rede virtual gerida pelo CMS e outro encontra-se numa rede externa ao *datacenter*. O Neutron é o componente que tem por missão gerir tudo o que está relacionado com a rede

no CMS OpenStack. Por outras palavras, é responsável por todo o *lifecycle* das redes virtuais presentes nas infraestruturas geridas pelo OpenStack. O ML2 é um *core plugin* que tem como objetivo suportar várias tecnologias para criação de redes virtuais num mesmo plugin L2. O OpenStack Neutron permite ainda o uso de *service plugins* que permite o acesso a funcionalidades específicas no contexto das suas funções de gestão de rede (como por exemplo o uso de uma firewal como *Firewall as a Service* (FaaS)). O OpenStack, o Neutron e o ML2 serão abordados mais à frente neste documento de forma mais exaustiva. Voltando ao L2-GW, este trata-se de um *service plugin* que permite ao Neutron em conjunto com o *core* plugin ML2, estender as redes virtuais presentes no *datacenter* para fora deste. O objetivo é atingido criando túneis VXLAN a partir das redes virtuais existentes na infraestrutura de rede gerida pelo Neutron e uma entidade localizada na rede externa que tem pelo nome de L2-GW, que durante os testes de desenvolvimento tomou forma num dispositivo de rede VTEP. Estes túneis criados são associados a um VLAN ID numa porta física no dispositivo VTEP, em que cada par túnel/VLAN ID está associado a uma rede virtual no *datacenter*. Como já foi dito anteriormente, a entidade L2-GW durante os testes da solução foi associado a um dispositivo hardware VTEP, mas não esta confinado só a essa hipótese. Foi criada uma API que permite o desenvolvimento de L2-GWs de outros tipos e usar outras tecnologias com por exemplo VLAN, GRE e outras.
A arquitetura de testes da solução é representada na figura 3.4.

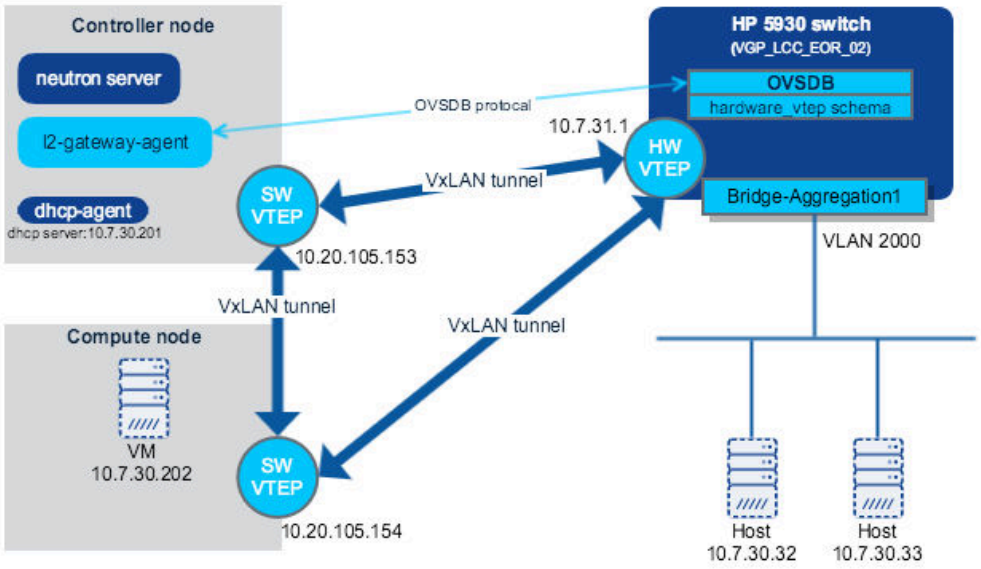

Figura 3.4: Exemplo de uso do L2-GW [25]

Esta abordagem proporciona uma solução bem integrada no CMS OpenStack, com uma interface de configuração bem definida ao mesmo tempo que, proporciona a possibilidade de ser usada nos mais variados tipos de cenários de rede. Isto porque, é fornecida uma API que permite a criação de L2-GW para esses cenários específicos e pelo facto de a interface de controlo ser simples e flexível, pois faz uso da API do Neutron para gestão e configuração dos L2-GWs.
Esta solução possui a vantagem de ter uma ótima integração com o componente de rede do OpenStack e o seu funcionamento interno, mas tem como desvantagem, como qualquer solução deste tipo, o problema de a mesma não poder ser utilizada nem portada de forma simples para outros CMSs.

3.6 ANÁLISE COMPARATIVA DAS SOLUÇÕES

Foram apresentadas neste capítulo cinco possíveis soluções, para a integração de redes *legacy* com as redes geridas por alguns tipos de software, que fazem parte da implementação do paradigma de *Cloud* num *datacenter*. Todas as soluções apresentadas possuem as suas vantagens e desvantagens, em que umas são mais direcionadas a resolver um determinado caso de uso e outras são mais abrangentes. A maioria das soluções apresentadas, foram desenhadas apenas para funcionar em conjunto com software específico, com recurso a um conjunto limitado de tecnologias de virtualização de rede. Outras permitem um espectro mais alargado a nível de personalização do seu funcionamento, mas sempre com a limitação ao seu uso no contexto específico de um software. Foi elaborada a seguinte tabela comparativa para melhor analisar as diferenças entre as soluções apresentadas neste capítulo:

Solução	**Plataforma suportada**	**Tecnologias de virtualização de rede suportadas**	**Protocolos suportados**	**Dispositivos suportados**
Colocar servidores físicos em redes virtuais	OVS	VXLAN	OVSDB	Dispositivos com suporte VTEP
LegacyFlow	OpenFlow	Redes programáveis (SDN)	OpenFlow	Dispositivos com suporte VLAN
Controlo de infraestrutura de rede para campus virtuais	Múltiplos CMS	VLAN e múltiplos tipos de redes *overlay* (GRE, VXLAN, etc.)	Pode suportar múltiplos	Dispositivos com suporte para as tecnologias de virtualização de rede suportadas pelo CMS
Estendendo as redes virtuais OpenStack - Neutron usando portas externas	OpenStack - Neutron	Redes *overlay* (GRE, VXLAN, etc.)	Pode suportar múltiplos	Qualquer dispositivo que suporte as redes *overlay* definidas no *deployment*
OpenStack Neutron - L2-GW	OpenStack - Neutron	VXLAN, mas podem ser usadas outras fazendo uso da API disponibilizada	OVSDB	Dispositivos com suporte VTEP

Tabela 3.1: Tabela comparativa das soluções

capítulo 4

Arquitetura da solução

Nos capítulos anteriores foi abordado o que já se conseguiu fazer nesta área, foi também introduzido ao leitor o background das tecnologias e definições mais importantes relativas a este trabalho. Neste capítulo são expostos os requisitos do sistema e os casos de uso que a framework deve suportar e que vão ditar quais as as melhores opções a tomar, tanto a nível de design como a nível de implementação, de modo a cumprir todos os objetivos.
Com os requisitos e os use cases identificados, é de seguida definido o design do sistema com base nestes. É através do design do sistema que se identificam e discutem os aspetos técnicos da solução, e onde se tem uma primeira noção dos possíveis desafios e como os contornar e resolver. Desta forma consegue-se encontrar a melhor caminho para uma implementação bem sucedida.

4.1 análise de requisitos

O requisito principal que a *framework* a ser desenvolvida terá de cumprir consiste em dotar os CMS da capacidade de estender as redes virtuais existentes no *datacenter* ao exterior deste. Ou seja, desenvolver uma *framework* que seja modular e o mais flexível possível, de modo a poder ser integrada em qualquer CMS, para os dotar de capacidades para configurar os dispositivos de rede exteriores à sua área de atuação, (área de atuação esta que normalmente é constituída por switches, firewalls e routers/switches virtuais que este consegue configurar numa instalação típica) de modo a criar ligações entre esses dispositivos e as suas redes virtuais. Isto para se poder alargar os domínios de colisão virtuais existentes no *datacenter*, para um ponto de acesso a dispositivos residentes no exterior. Estas extensões às redes virtuais deveram ser criadas de forma automática, ou seja, a *framework* deverá ser responsável por analisar a topologia da rede externa e aplicar as extensões de acordo com regras definidas pelo administrador para escolha do melhor caminho na rede.
A *framework* terá de ser idealizada para ser o mais simples e flexível possível, e não pode de forma alguma, estar dependente de uma única plataforma e/ou CMS. A mais importante decisão a tomar será sobre a forma como a *framework* se irá integrar com o CMS. Existem duas hipóteses possíveis, desenvolver uma solução *standalone*, ou um conjunto de módulos que podem ser integrados nos componentes dos CMS. Para fazer uma avaliação correta é necessário ter em mente todos os prós e contras de ter uma solução *standalone*, ou uma solução que seja integrável com os CMSs. Se se

optar por uma solução *standalone*, esta poderá disponibilizar uma API em que o CMS apenas terá de fazer pedidos para essa API, e a solução é que possui toda a lógica de criação das extensões de rede. Terá também de aplicar as configurações nos dispositivos para criação dos links, e monitorização dos dispositivos da rede externa onde foram criadas as extensões. O esforço de desenvolvimento neste caso é maior ao inicio, mas a facilidade de integração com os CMS é grande. No entanto esta abordagem possui vários problemas como a possível duplicação de dados (por exemplo duplicação de dados acerca da rede, e dos utilizadores da *Cloud*) e duplicação de funcionalidades (por exemplo API e gestão de processos).

Por outro lado, a solução ao ser criada de modo a dotar os CMS de capacidades para estender as suas redes virtuais, resolve parte das principais desvantagens de uma possível solução *standalone*. Ou seja, torna possível o uso da API e da gestão da base de dados normalmente disponíveis em todos os CMS, muitos destes até proporcionam meios que permitem uma maior facilidade em adicionar novas funcionalidades a estes recursos. Tirando partido destas faculdades, é possível reduzir ao inicio o tempo de desenvolvimento por não ser necessário criar novas APIs, nem fazer a lógica de gestão da base de dados. Desta forma é possível alcançar uma maior eficiência nas operações a realizar, devido a uma melhor integração com os CMSs. Contudo, esta aproximação também traz alguns problemas, como a necessidade de um maior esforço de integração com os diversos CMS, pois cada um deles implementa as suas APIs e base de dados de forma diferente. Isto vai obrigar a um maior esforço aquando da integração da *framework* num CMS específico.

Embora pese o facto mencionado anteriormente de que a *framework* ao ter de ser o mais simples possível, esta terá também de interoperar da melhor forma possível com o CMS, e ao mesmo tempo fazer uso o mais possível dos funcionalidades disponibilizadas por este. Ou seja, a *framework* deverá tirar partido das extensões e outros recursos (bases de dados, API, etc.) existentes nos CMS, para evitar a duplicação de informação e evitar estar a implementar o que já foi implementado (duplicação de código). A *framework* terá também de fornecer meios para ser possível a sua fácil integração ao CMS e não o contrário. Terá ainda de ser adaptável ao funcionamento interno da gestão das redes virtuais por parte do CMS em questão, isto para seja possível criar pontos de ancoragem para as extensões destas na rede externa. Desta forma, é possível verificar que a *framework* além de aplicar configurações nos dispositivos físicos, terá também de o fazer se necessário nos dispositivos virtuais geridos pelo CMS que interligam as VMs nos *compute nodes*. Mais especificamente, no que concerne à gestão dos dispositivos da rede externa, a *framework* deverá ser capaz de fornecer meios para que os administradores da *Cloud*/rede, possam fazer com que qualquer dispositivo da rede possa ser configurado pela mesma.

Todas as configurações aplicadas pela *framework* terão, obviamente de ser aplicadas de forma não disruptiva, isto é, as novas configurações aplicadas pela *framework* não podem de forma alguma modificar/apagar as configurações já presentes nos dispositivos de rede. As extensões criadas pela *framework* são constituídas por *links* criados entre dois dispositivos em cada segmento da rede, em que estes *links* podem ser de um tipo qualquer de rede virtual (por exemplo VLAN, GRE, VXLAN, etc.) desde que seja suportado por ambos. Estes *links* fazem parte de um *path*, que pode ser constituído por um ou mais *links*. Em cada *path* está associado a uma rede virtual gerida pelo CMS. Por exemplo, na figura 4.1 os *links* que estão ilustrados a amarelo pertencem todos à mesma rede virtual, logo formam um *path* para essa rede. Os *links* a vermelho cinza, fazem parte de um *path* de outra rede virtual diferente.

A *framework* deverá fornecer, com já foi referenciado acima, um meio para automatizar as tarefas de criação destes *path*. Para isso deverá recolher toda a informação da topologia da rede externa de que necessita, isto é, recolher a informação acerca dos dispositivos que a compõem (por exemplo

informações relacionadas com as interfaces, o seu tipo, estado). Quando se pretende fazer uma extensão a uma determinada rede virtual gerida pelo CMS, a *framework* deverá também identificar o melhor caminho na topologia até ao dispositivo de rede que se pretende que dar acesso, recorrendo a algoritmos ou regras definidas pelo administrador. A informação acerca da topologia deve também ser atualizada de forma regular, devido à possibilidade de os dispositivos deixarem de estar disponíveis. Devido a este último facto, pode haver a necessidade de refazer alguns *paths*, criando novos *links* noutros segmentos. Isto pode acontecer quer por avaria, quer por reorganização da topologia, podendo haver a necessidade de refazer alguns *paths* dos links já criados. Para isso a *framework* deverá fornecer mecanismos de monitorização dos dispositivos que devem verificar o seu estado de acordo com o políticas definidas pelo administrador. Isto poderá ser feito, no mais simples dos casos, utilizando o protocolo *Internet Control Message Protocol* (ICMP) ou o protocolo SNMP.

Com todas estas considerações, a solução deverá ter se possível o melhor dos dois mundos, isto é, deve aproveitar as possibilidades de extensão das API e base de dados disponíveis nos CMS e deverá ser independente na forma como aplica as configurações nos dispositivos para criar as extensões (*paths*) às redes virtuais. Esta independência trará também a hipótese de ter outros mecanismos de gestão a gerir a rede externa sem ser o CMS. A *framework* deverá fornecer interfaces genéricas para facilitar a integração com as API dos diversos CMSs, e ter formas de se adaptar às funcionalidades presentes deste que possam ser utilizadas para agilizar o *deployment* da *framework* no CMS (por exemplo a gestão de processos de forma individual). Esta integração permite-nos ter ganhos a nível de eficiência devido a uma melhor integração com o CMS nas operações de configuração dos dispositivos externos entre outras.

Em suma, o principal objetivo é desenvolver uma *framework* que dote os CMS das necessárias capacidades para poder estender as redes virtuais criadas e geridas por si, usando ao máximo as capacidades de extensão das funcionalidades já existentes, sem que isto torne a *framework* demasiado dependente de um CMS em específico. A *framework* deve ser modular para se poder adaptar da melhor forma possível ao CMS e permitir que seja fácil a configuração e monitorização de qualquer tipo de dispositivo existente na rede *legacy*.

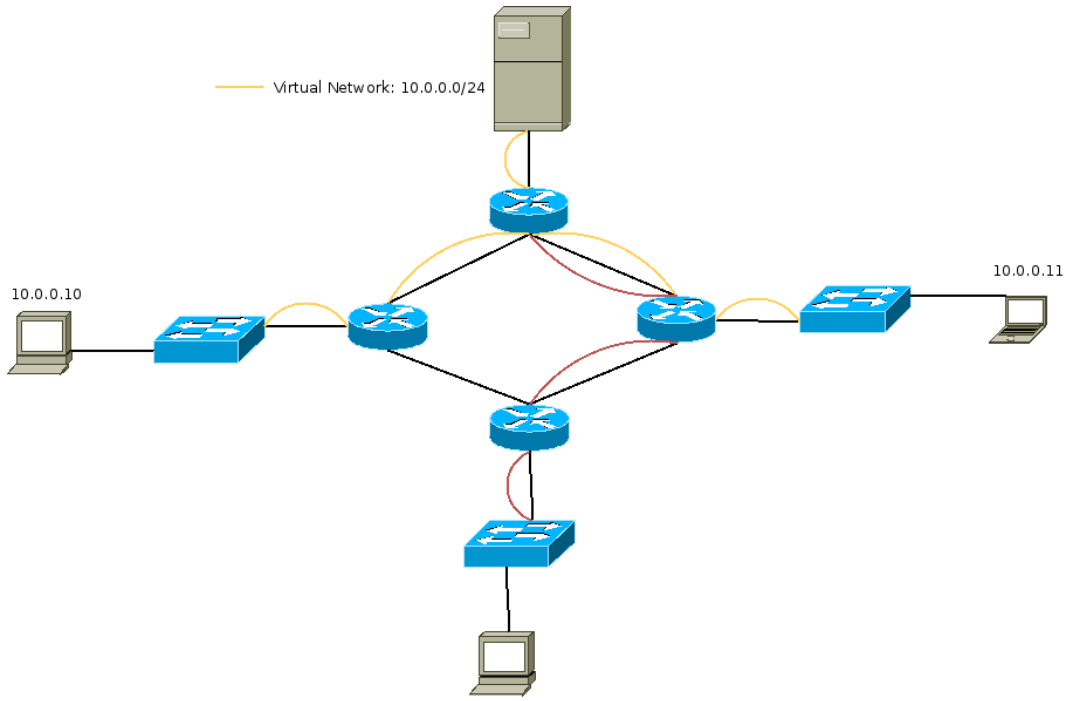

Figura 4.1: Figura ilustrativa dos *path* criados na rede externa.

4.2 CASOS DE USO

Nesta secção serão apresentados alguns casos de uso que a solução deverá cumprir.

4.2.1 CASO DE USO RELATIVO À UNIVERSIDADE DE AVEIRO

A Universidade de Aveiro é constituída por 32 departamentos suportados por 65 edifícios numa área equivalente a 92 campos de futebol e por 2 polos fora da cidade[1]. O número de colaboradores (professores e auxiliares técnicos) e estudantes é de aproximadamente 16000 pessoas, e para dar conta das necessidades deste número de pessoas existem aproximadamente 4800 computadores distribuídos pelos departamentos e edifícios de gestão. Para tornar a comunicação entre estes possível são usadas ligações de fibra ótica para interligar os departamentos e Ethernet para as ligações no interior dos departamentos. Com estes números já generosos e com uma comunidade muito ativa em investigação e desenvolvimento, tanto dentro da instituição como em colaboração com outras instituições, os serviços de IT devem estar sempre a funcionar nas melhores condições possíveis de fiabilidade e disponibilidade, para poderem atender às exigentes necessidades de uma comunidade deste tipo.
Para garantir que os serviços IT estão sempre com um grau de disponibilidade elevado, existe uma entidade responsável pelo departamento das IT na universidade chamado Serviços de Tecnologias de Informação e Comunicação (STIC). Os STIC possuem entre mãos um grande número de serviços a seu cuidado, que necessitam de garantir o seu bom funcionamento com grande grau de disponibilidade. Adicionando a este árduo trabalho, têm também de lidar com as constantes mudanças nos requisitos

[1]Dados do site da UA: `http://www.ua.pt/campusdaua`

destes, e a constante chegada e partida de pessoas e dispositivos que necessitam de usufruir da rede e consequentemente dos serviços disponíveis. O acesso à rede e seus serviços é feito por cabo Ethernet ou Wi-Fi.
Todo o trabalho de garantir o bom funcionamento da estrutura é feito pelo seu pessoal que, devido à dimensão do campus, é uma tarefa muito complicada de se fazer. Como consequência disso, parâmetros como a eficiência, disponibilidade e rapidez dos serviços podem ser postos em causa. Um exemplo destas tarefas, é por vezes haver a necessidade de configurar um *setup* de rede específico, numa sala de aula onde um exame irá decorrer. Os colaboradores dos STIC, necessitam de configurar a rede de forma a que seja criado o ambiente necessário para que exame possa decorrer. Para isso, os colaboradores têm de percorrer dispositivo a dispositivo, para aplicar as configurações necessárias para atingir esse objetivo. Para poder libertar os STIC desta difícil rotina, o passo seguinte será adotar uma solução que seja baseada em *Cloud*, isto leva a que se possa ter uma solução de gestão dos recursos IT centralizada, e logo mais simples e eficiente. As vantagens são muitas, entre elas a possibilidade de fazer as configurações mencionadas no exemplo anterior, de forma centralizada e com menor esforço. Para além disso, possibilita a capacidade de gerir a infraestrutura remotamente (por exemplo através de um portátil ou de um *smartphone*), o que por consequência reduz o tempo de resposta a possíveis complicações que possam surgir, ao permitir que estas possam ser resolvidas de forma remota.
Mas, para que a solução baseada na *Cloud* seja eficaz, esta terá de ter a capacidade de gerir a parte da rede física já existente e não somente o *datacenter*. Esta capacidade não foi implementada de forma eficaz nos CMS, pelo que uma solução de *Cloud* hoje ainda não é viável para uso neste caso da universidade. Não é viável neste caso, porque tendo em conta a organização da infraestrutura IT, os vários departamentos estão dispersos e com muito equipamento com diferentes características. Ora, por norma uma solução baseada em *Cloud*, nos dias de hoje, está preparada apenas para gerir um conjunto de recursos específicos e homogéneos que estão localizados num *datacenter*, ou seja, centralizados num ponto. Ao implementar uma solução baseada em *Cloud* num cenário como o da Universidade de Aveiro, é de todo o interesse que a nova infraestrutura consiga também fazer a gestão do equipamento mais antigo já existente, o que hoje tal não é possível nas soluções de *Cloud* existentes sem um grande esforço por parte dos administradores IT.

4.2.2 PERMITIR O PARADIGMA DE FOG COMPUTING EM AMBIENTES IoT

Como já foi referido na secção 2.1.5 onde foi exposta a definição de *Fog Computing*, este é um conceito particularmente importante na área de IoT. A solução proposta irá permitir uma forma integração entre a *Cloud* e o paradigma *Fog Computing*, possibilitando a colocação de dispositivos externos no mesmo domínio de colisão, que as VMs residentes num *datacenter* por meio da extensão das redes virtuais que as interligam. Esta funcionalidade encaixa perfeitamente na definição de *Fog Computing* que defende que os dispositivos que estão mais próximos dos utilizadores finais da rede, sejam mais pró-ativos na gestão e processamento dos dados [26]. Com a solução proposta neste trabalho é preenchido precisamente esse paradigma, ao possibilitar que esses dispositivos na periferia da rede sejam colocados na mesma rede que os servidores virtuais no *datacenter*. Como consequência disso, é possível ter uma melhor gestão dos dados e sua análise e abrir novas portas para novos serviços e melhorar os já existentes.
Por a solução fornecer uma forma que permite a adoção do paradigma de *Fog Computing*, é possível

também afirmar que esta solução pode igualmente ser muito útil no desenvolvimento de novas soluções baseadas em IoT, isto porque vai proporcionar uma melhor integração das capacidades de gestão, processamento e armazenamento em ambientes IoT com o *datacenter* onde estarão disponíveis os principais meios de processamento de armazenamento. Um exemplo desta integração é a possibilidade de ter um ou mais dispositivos a recolher dados em localizações diferentes fora do *datacenter*, onde existe a necessidade dos dados recolhidos serem processados por um servidor que está alojado no *datacenter*, esta solução possibilita o acesso L2 entre esses dispositivos e o servidor.

4.2.3 SERVIDORES PARA APLICAÇÕES ESPECÍFICAS

Alguns tipos de tarefas necessitam de requisitos especiais que os servidores virtuais não conseguem cumprir pelos simples facto de serem "virtuais", esta característica limita-os no acesso a I/O e consequentemente ao uso de aplicações que necessitem de acesso direto ao hardware. Com a solução proposta será possível dar a acesso a um conjunto de máquinas virtuais, a uma máquina ou a um conjunto delas que se encontram fora do *datacenter*, em que estas máquinas possuem hardware e software específico, para tarefas cujos os dados são necessários nos servidores virtuais no *datacenter*. Por exemplo, é possível ter uma máquina fora do *datacenter* com um *Graphics Processing Unit* (GPU) com alta capacidade, e dar-lhe acesso por rede L2 a um conjunto de servidores residentes no *datacenter*. A solução irá permitir criar um ponto de acesso no dispositivo de rede mais próximo da máquina externa, e criar um *path* de forma a colocar todas as maquinas no mesmo domínio de colisão. O utilizador terá apenas de pedir a porta através do CMS, para que a *framework* trate de todas as configurações necessárias para atingir esse objetivo, isto de forma totalmente transparente para o utilizador. Esta capacidade trará vantagens às empresas que pretendem mover para a sua infraestrutura IT, para uma solução baseada em *Cloud*. Mas que por diversos motivos, ainda estão de alguma forma limitados devido a estes casos de uso específicos, em que a sua funcionalidade não permite a virtualização das máquinas.

4.2.4 EXTENSÃO PARA OUTRO *datacenter*

Em muitas empresas é complicado prever de que forma irá ser o seu volume de negócios, consequentemente torna-se difícil de prever a necessidade de recursos de IT. Muitas das empresas bem sucedidas quando chegam a um certo patamar de volume de negócios, tendem a abrir novos polos/novos escritórios de forma a expandir as suas operações. Devido a isto, por vezes é necessário construir uma nova infraestrutura nesses novos escritórios, quer por vezes devido à distância entre eles ou devido ao volume de operações nos vários polos entre outras razões. Existem atualmente soluções que permitem fazer essa interligação tanto em L2 como em L3, um exemplo comum são as VPN em L3. No entanto, com esta solução, é possível criar ligações entre vários segmentos de rede diferentes, usando vários tipos de tecnologias. Um exemplo deste caso, pode ser a necessidade de configurar um *Multi-Protocol Label Switching* (MPLS) *overlay* nos dispositivos ao longo de uma rota, para uma interligar *datacenters* em localizações diferentes.

4.3 DESENHO DA SOLUÇÃO

Com todos os requisitos do sistema e casos de uso identificados, é possível definir na estrutura da solução. Foi identificado que a solução terá de ser modular e promover a flexibilidade, para permitir a sua simples integração com os vários CMS. Terá também de ser autónoma nalgumas operações, como por exemplo, inteirar-se acerca da topologia da rede externa e criar automaticamente *links* virtuais e portas para acesso às redes virtuais no *datacenter*. Para alcançar este objetivo de obter uma solução modular, é imperativo criar uma abstração da rede externa para melhor lidar com os aspetos relacionados com a informação necessária. Isto para que a *framework* seja capaz de gerir e aplicar as configurações necessárias nos dispositivos de rede externos.

4.3.1 REPRESENTAÇÃO DOS DISPOSITIVOS DE REDE

Para melhor alcançar os objetivos, é necessário ter em mente o grau de complexidade com que se está a lidar, que neste caso ao serem redes físicas podem chegar a grandes níveis de complexidade. Estas são compostas por um conjunto alargado de dispositivos, muitas das vezes completamente diferentes uns dos outros, tais como routers, switches, hubs, *Virtual Switch*s (vSwitchs), *Access Point*s (APs), firewalls entre outros. Mais ainda, muitos deles são muitas das vezes provenientes de diferentes marcas e modelos com diferentes características. É necessário também de ter em conta a forma como algumas topologias de rede são projetadas, que por vezes devido a limitações de orçamento aquando da sua implementação, não serão as ideais.
Ao observar com cuidado os requisitos, é possível verificar que não é necessário lidar com toda essa complexidade existente nas redes físicas. Não é necessário saber muitas das características específicas de cada dispositivo existente na rede, é apenas necessário saber como chegar aos dispositivos, quais são os seus vizinhos e ter alguma informação importante acerca das suas interfaces. Isto para ser possível aplicar as configurações necessárias para criar *links*, através de uma das tecnologias *overlay* suportadas (por exemplo VLAN, GRE ou outras), para poder chegar ao objetivo de estender a rede virtual para a localização desejada.
Para melhor atingir esse objetivo, depois de alguma análise optou-se por seguir como base, o mapeamento de rede idealizado pelo Filipe Manco na sua dissertação de mestrado [22], e já mencionado no capítulo 3.3. A abstração da rede física proposta, foi desenvolvida tendo em mente o objetivo de simplificar, a criação e a gestão de redes virtuais em redes físicas, extraindo para isso da rede física apenas a informação necessária para atingir esse objetivo [22]. Com esta abstração, é possível descrever completamente a rede física e as redes virtuais nela criadas, e ter a informação necessária para configurar os dispositivos quando necessário.
Esta abstração é constituída pelas quatro entidades descritas abaixo:

- **ExtNode** - Um `ExtNode` representa um dispositivo que pode ser configurado pela *framework*, em que este pode ser físico ou virtual. Os `ExtNodes` pode tomar forma em diversos tipos de dispositivos, podem ser um switch, um router, ou um vSwitch entre outros. As principais funções dos `ExtNodes` são fornecer `ExtPorts`, criar e interligar `ExtLinks`. Os `ExtNodes` são ligados entre si usando `ExtSegments`.
- **ExtPort** - Uma `ExtPort` representa um ponto de acesso para uma rede virtual. Através desta, os dispositivos a esta ligados podem enviar e receber dados para as redes virtuais de forma

isolada. A entidade `ExtPort` possui o mesmo tipo de abstração tal como a entidade `ExtNode` e portanto pode assumir várias formas, por exemplo um `ExtNode` do tipo AP pode ter um SSID que fornece acesso a uma rede virtual específica, o SSID no ponto de vista da *framework* neste caso é uma `ExtPort`.

- **ExtSegment** - Um `ExtSegment` representa um domínio no qual é possível criar `ExtLinks`, em que cada um destes possui um ou mais tipos de de redes *overlay* suportadas para criação de `ExtLinks`. Um `ExtSegment` possui `ExtNodes` que atuam como *endpoints* para os `ExtLinks`. Um `ExtSegment` pode ou não mapear diretamente um segmento físico. No caso de o mesmo não mapear um segmento físico, este tem de suportar `ExtLinks` *overlay* baseados em túneis. Este tipo de `ExtSegment` pode ser por exemplo a Internet.
- **ExtLink** - Um `ExtLink` é uma ligação *overlay* que é criada num `ExtSegment`. Esta ligação virtual é feita com recurso a tecnologias (VLAN, GRE, VXLAN, entre outras) suportadas pelo `ExtSegment`. Por outras palavras, a entidade `ExtLink` representa uma ligação virtual num determinado `ExtSegment` que possui dois `ExtNodes` como *endpoints*. Os `ExtNodes` têm que suportar a tecnologia *overlay* necessária para criar o `ExtLink`. Os `ExtLinks` estão sempre associados a uma rede virtual presente no CMS.

Esta representação abstrata da rede física deverá ser mantida na base de dados dos CMS e gerida pela *framework*, para rápido acesso à mesma e posterior agilização das operações relacionadas com a *framework*. A gestão das operações relacionadas com estas entidades será sempre efetuada pela *framework* com recurso às ferramentas disponibilizadas pelo CMS sempre que possível.

4.3.2 ORGANIZAÇÃO INTERNA DA SOLUÇÃO

Depois de identificar as entidades que nos permitem abstrair a complexidade da rede física, o foco é agora colocado na forma de como vai ser efetuada a organização interna da arquitetura da solução. Para uma melhor gestão do *workflow* da *framework*, esta será dividida num conjunto de módulos, em que cada um destes terá a seu encargo um conjunto de funcionalidades bem definidas no seu âmbito. Com esta aproximação modular da solução os ganhos são significativos na interoperabilidade da solução com os CMS. Como consequência, é possível uma melhor integração com estes e uma melhor abordagem na implementação da solução, identificando de uma forma mais eficaz as funcionalidades pretendidas. Desta forma, tal como na programação orientada a objetos, ao ter uma solução modular que por sua vez os seus módulos podem ser reutilizados por outras aplicações. Consegue-se também promover ao mesmo tempo uma melhor organização interna da solução, permitindo que cada módulo tenha uma tarefa a seu encargo bem definida. Com vista a atingir esse objetivo de obter uma solução modular, procedeu-se à divisão dos mesmos da seguinte forma. Um módulo chamado de *Network Controller* que possui os seguintes submódulos: *Topology Discovery*, *Network Mapping* e *Device Controller Manager* este último pode ter vários submódulos a ele ligados que são do tipo *Device Controller*. Esta organização dos módulos está representada na figura 4.2.
São agora a seguir, enumeradas as principais características e funcionalidades de cada um dos módulos.

- **Network Controller** - O *Network Controller* é o *core* da *framework* e só deverá existir um por *deployment*. Este módulo é responsável pelo funcionamento base da solução, ou seja, é ele que faz toda a orquestração dos vários módulos por forma a obter a funcionalidade pretendida da

framework. Tem a seu encargo a gestão dos pedidos relativos às entidades de abstração de rede, apresentadas anteriormente, feitos pela API do CMS e posterior salvaguarda da informação na base de dados. Ou seja, é responsável pela interação com os mecanismos de persistência e API do CMS. Tal como já foi mencionado, é também responsável pela interação com os outros módulos relativos à solução, isto é, por exemplo pedir uma pesquisa à topologia ao módulo de *Topology Discovery*, ou interagir com o *Device Controller Manager* de forma a aplicar configurações nos dispositivos ou receber informações de monitorização dos mesmos.

- **Topology Discovery** - Este módulo tem como objetivo pesquisar e recolher as informações necessárias para que, a *framework* possa estender as redes virtuais até um determinado dispositivo na rede externa. Deve ser utilizado pelo *Network Controller* para fazer uma pesquisa à topologia sempre que este não possua informações acerca de um dispositivo e respetiva interface. Se o dispositivo figurar na base de dados mas não estiver disponível, ou se for alertado pelo *Device Controller Manager* que este está indisponível (por avaria, remoção do dispositivo da rede, etc.) e é necessária fazer a sua remoção da base de dados. No caso deste módulo, a *framework* disponibiliza uma API por forma a que o operador possa implementar a solução que mais lhe convém, isto é, que tecnologia a usar para obter a informação relativa aos dispositivos (por ex.: SNMP, telnet, SSH, ou outras).
- **Network Mapper** - O módulo de *Network Mapper*, tem como tarefa encontrar um *path* na rede até uma determinada localização, usando para isso um conjunto de regras definidas pelo administrador aquando da instalação da *framework*. Este *path*, tem como *endpoint* uma interface num dispositivo na rede externa, que vai permitir o acesso a uma determinada rede virtual. Tal como no módulo anterior de *Topology Discovery* é disponibilizada uma API que permite ao operador do CMS, implementar a solução de mapeamento que pretender com as suas devidas configurações. É função do *Network Mapper* fornecer ao *Network Controller*, um *path* para um dispositivo externo quando assim o é pedido.
- **Device Controller Manager** - Este módulo tem como tarefa, fazer pedidos de configuração aos diversos *Device Controllers* a ele associados. Este módulo serve como mediador entre os *Network Controller* e os *Device Controllers*, isto porque os *Device Controllers* podem possuir interfaces de comunicação diferentes uns dos outros, por exemplo um *Device Controller* pode ter uma interface que comunique por *Remote Procedure Call* (RPC) e outro pode comunicar por *webservices*.
- **Device Controller** - O *Device Controller* tem como objetivo receber pedidos de configurações do *Device Controller Manager* e aplicá-los ao respetivo dispositivo, usando para isso um *Device Driver* que possui os meios necessários para a aplicação das configurações num determinado tipo de dispositivo. Ou seja, a sua função é receber pedidos de configuração, carregar o *Device Driver* correspondente a cada dispositivo e aplicar as configurações. Um *Device Controller* pode ter a seu encargo mais do que um dispositivo da rede externa.
- **Device Driver** - Um *Device Driver* possui toda lógica para fazer a ligação e aplicar as configurações a um dispositivo específico, fazendo uso para isso, de uma interface suportada (por exemplo CLI ou SNMP) por este.

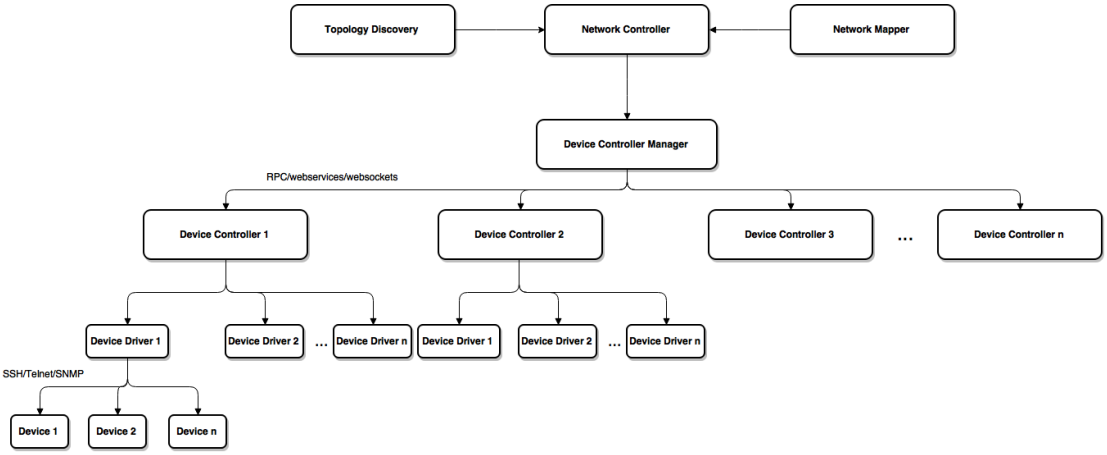

Figura 4.2: Organização dos módulos da *framework*

4.3.3 OPERAÇÕES RELATIVAS ÀS ENTIDADES DE ABSTRAÇÃO DE REDE

A *framework* deverá facultar aos CMSs meios para suportar as operações na rede, relativas às quatro entidades mencionadas anteriormente. As operações relativas à gestão dos dados necessários ao funcionamento da *framework*, deverão ser realizadas com recurso a meios já existentes para tal no CMS. Isto é, como já foi referido neste capítulo, esta gestão deverá ser feita com recurso à API e ao gestor da base de dados interna do CMS, usando para isso as extensões que são disponibilizadas para esse efeito. Com isto está-se a assumir que a *framework* deverá ser usada em CMSs com capacidades de extensão dos seus modelos de dados, e com uma API de gestão da *Cloud* igualmente extensível.
O CMS deverá implementar as operações *Create, Read, Update and Delete* (CRUD) que podem despoletar operações tanto a nível da base de dados, tanto a nível de operações nos dispositivos de rede externos consoante o tipo de operação e o tipo de entidade. As entidades que abstraem a rede física, podem também ser alvo de operações CRUD quando são efetuadas ações relacionadas com a procura de novos dispositivos na rede externa, e na criação automática de caminhos para estender as redes virtuais.
Os utilizadores que possuam as devidas permissões poderão criar portas num dispositivo pertencente à rede externa. Aquando do pedido dessas portas a *framework*, terá de criar os respetivos links nos segmentos de acordo com o definido pelo *Network Mapper*, e configurar a respetiva porta de acesso. O processo de criação do *link* associado a um segmento só é despoletado, se ainda não existir nenhum *link* associado à rede virtual pretendida nesse mesmo segmento. O processo de eliminação de uma porta externa, segue os mesmos passos do anterior só que de forma inversa. Os *links* serão apenas eliminados dos segmentos, caso esses mesmos *links* não estejam associados a mais nenhuma porta externa. A seguir é mostrado um diagrama de atividades, que mostra a criação de uma porta externa pela *framework*.

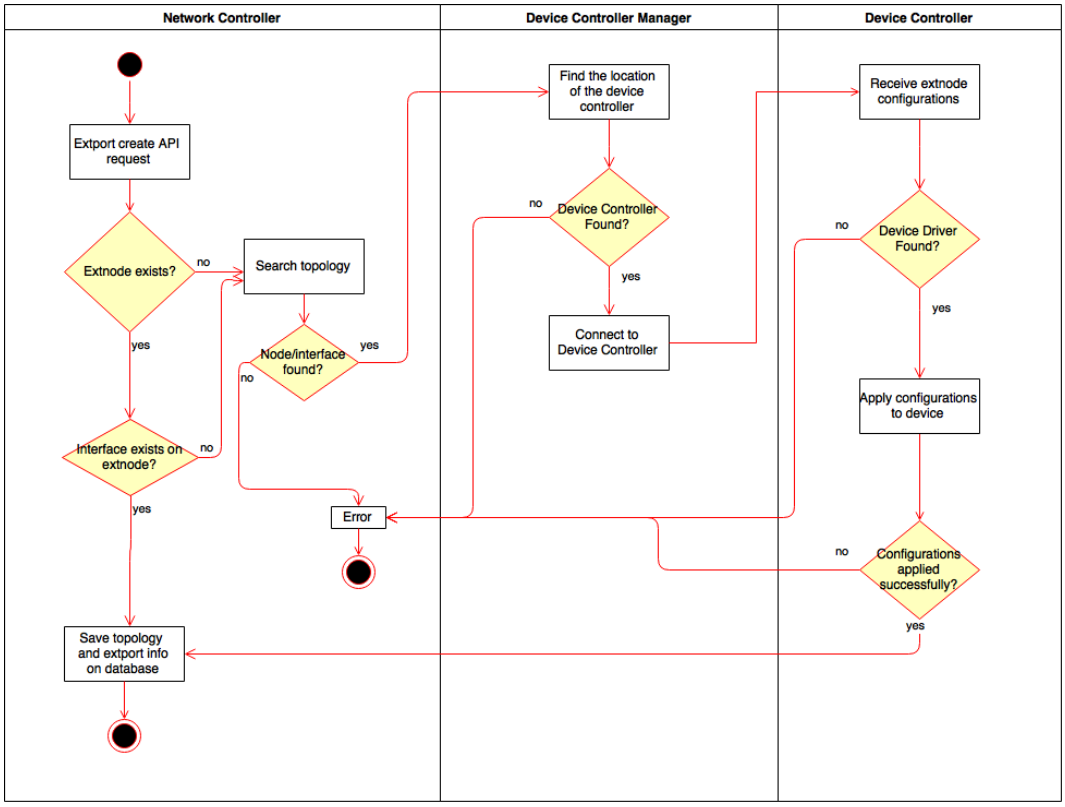

Figura 4.3: Diagrama de atividades representativo da criação de uma porta externa

capítulo 5

Implementação da Solução

Neste capítulo é abordada a forma como a arquitetura apresentada no capítulo anterior é integrada num CMS. O CMS escolhido foi o OpenStack, por ter a vantagem de ser facilmente extensível e ser um software FOSS. Pesa também o facto, de este ser um dos softwares de Cloud IaaS mais usado na atualidade. É primeiramente feita uma abordagem geral à arquitetura de funcionamento do OpenStack. De seguida, é abordado de forma mais aprofundada o componente responsável pela gestão da rede o Neutron, que neste caso vai ter especial importância por ser onde vai ser integrada a framework. Posto isto, é dado a conhecer o setup usado para a implementação da solução e respetivas configurações. Por último, é exposta a forma que foi adotada para integrar a framework no Neutron.

5.1 Análise detalhada ao OpenStack

5.1.1 Arquitetura da rede

Na figura 2.2 é possível observar os componentes e respetivos serviços mínimos para obter uma *Cloud* baseada em OpenStack. Importa também abordar de que forma estes componentes são interligados entre si. Por norma, e para atingir um grau de fiabilidade e disponibilidade mais elevado, as ligações entre os diversos componentes são constituídas por várias redes distintas, de modo a fazer uma separação do tráfego pelo seu tipo e características. Um cenário simples e recomendado para um *deployment* OpenStack que cumpre os requisitos mínimos a nível de fiabilidade e disponibilidade é constituído pelas seguintes redes e é demonstrado na figura 5.1.

- **Management network** - Esta rede é dedicada para a comunicação entre os diversos componentes do OpenStack. Nesta rede deve ser garantido que todas as máquinas que alojam os diversos componentes, estejam acessíveis apenas no *datacenter* por questões de segurança.
- **Guest network** - Esta rede é destinada a suportar o tráfego das redes virtuais criadas pelos utilizadores da *Cloud*, ou seja, é a rede por onde fluí todo o tráfego das VMs. Os requisitos a nível de endereçamento IP desta rede dependem do *Core Plugin* configurado no Neutron, e das configurações feitas pelos utilizadores aquando da criação das suas redes virtuais.

- **External network** - Esta rede fornece acesso externo às redes virtuais dos utilizadores, isto é, permite dar acesso à Internet através de uma NAT ou através de associações entre endereços IP externos e endereços IP de redes virtuais (*Stateful NAT* (SNAT)), permitindo assim o acesso às VMs a partir do exterior. Estes endereços externos são disponibilizados ao *Network Node* que gere essas associações. Os IPs disponíveis nesta rede devem estar acessíveis pela Internet.
- **API network** - Esta rede é destinada aos pedidos para a API do OpenStack. Esta rede deverá ser acessível do exterior, é por ela que chegam todos os pedidos relacionados com a gestão da *Cloud* OpenStack. Ao colocar os pedidos de gestão à API numa rede separada, consegue-se obter uma grau mais elevado de segurança na infraestrutura *Cloud*.

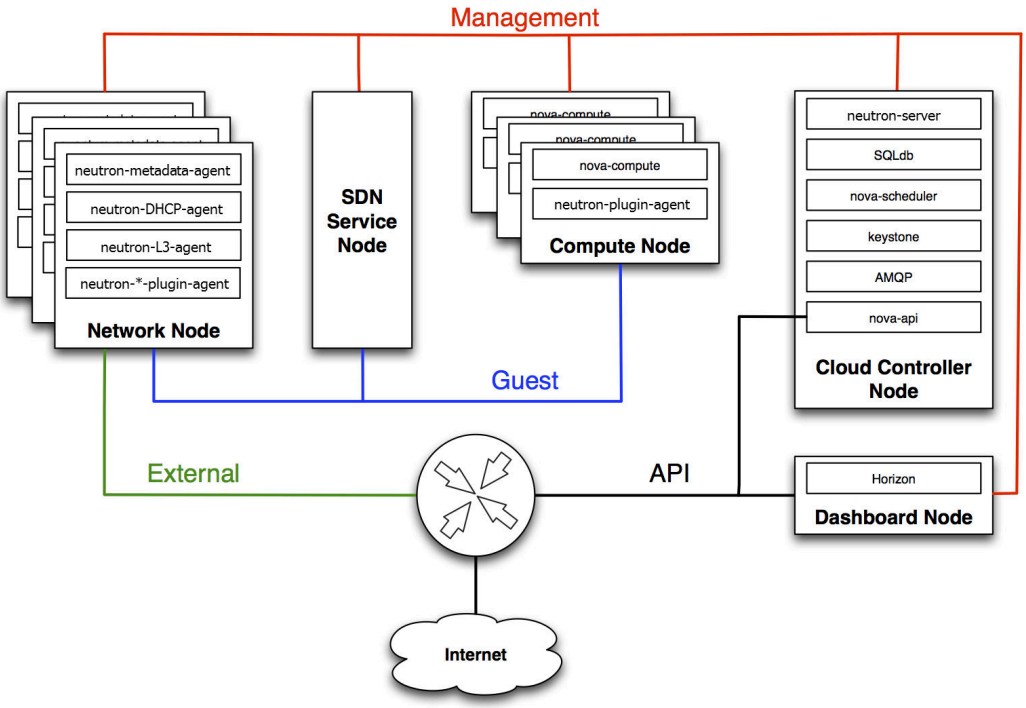

Figura 5.1: Arquitetura de rede simples de uma *Cloud* baseada em OpenStack [27]

5.1.2 NEUTRON

O Neutron, como já foi referido anteriormente, é o componente do OpenStack destinado a gerir todas as operações relacionadas com as redes no ambiente *Cloud* OpenStack. É um componente *standalone*, que possui a capacidade de permitir aos utilizadores, a criação de topologias de rede complexas sem limites impostos ao uso de elementos de rede (falta de portas, falta de dispositivos, etc.). Isto tudo de uma forma totalmente isolada para cada utilizador.
O Neutron possui um conjunto de entidades base que lhe permite gerir as suas redes virtuais, essas entidades são *Networks*, *Subnets* e *Ports*. No que concerne a persistência destas entidades, é usado um gestor de base dados *Object-Relational Mapping* (ORM) que neste caso é SQLAlchemy[1], que permite

[1]http://www.sqlalchemy.org

a gestão destas entidades perante a base de dados usando objetos Python. Por usar o SQLAlchemy o Neutron pode ser configurado para usar os mais diversos mecanismos de gestão de base de dados. Por defeito, o usado por todos os componentes do OpenStack numa instalação típica é o MySQL[2]. Os pedidos de operações no Neutron podem ser feitos diretamente através da sua API RESTful, através de um cliente CLI que interage com a sua API ou ainda através do *dashboard* web Horizon. O Neutron é capaz de orquestrar as redes virtuais de várias formas, usando para isso várias tecnologias entre elas estão VLAN, GRE e VXLAN. Possui a capacidade de criar routers virtuais, de forma a tornar ágil o acesso das redes internas dos utilizadores a redes externas, sendo possível o acesso através de NAT ou *floating IPs*.

Na sua génese de funcionamento é constituído por dois grandes módulos que podem ser executados em diferentes nós de computação, o *Neutron Server* e os *Neutron Agents*, estes módulos interagem entre eles por meio de *Remote Procedure Call* (RPC). O mecanismo que implementa um RPC no Openstack tem o Nome de Oslo RPC, e é capaz de usar as mais conhecidas implementações de *brokers* de mensagens que usam o protocolo *Advanced Message Queuing Protocol* (AMQP). O *broker* de mensagens usado por defeito pelo Oslo RPC é o RabbitMQ[3]. O Neutron possui um *Core Plugin* que implementa todas as operações relacionadas com as entidades base do Neutron existentes na sua RESTful API (*Networks*, *Subnets* e *Ports*), e pode também implementar operações específicas de uma extensão à RESTful API. As extensões suportadas por cada *Plugin* são descritas no ficheiro de configuração do Neutron. Existem também os *Neutron Agents* que normalmente são processos que são colocados a correr em nós, onde o Neutron necessita de aplicar configurações em serviços de rede a correr nesse mesmo nó. Essas configurações têm de ser efetuadas *on-site* para atingir uma determinada configuração de rede. Os agentes mais utilizados são o DHCP *Agent* que faz a gestão dos IPs nas redes virtuais, orquestrando o servidor de DHCP instalado na máquina onde está a ser executado. Existe também o L3 *Agent* que faz a gestão dos routers virtuais. E por fim o *Plugin Agent*, que tem como objetivo aplicar as configurações necessárias nos dispositivos de rede virtuais existentes nos nós, de forma obter um determinado cenário de rede. Por outras palavras, o *Plugin Agent* funciona como um intermediário perante o *Core Plugin* que está em execução no Neutron Server.

Um *Core Plugin* deve implementar todas as operações relacionadas com as entidades base do Neutron existentes na sua RESTful API (*Networks*, *Subnets* e *Ports*), e pode também implementar operações específicas de uma extensão à RESTful API. As extensões suportadas por cada *Plugin* são descritas no ficheiro de configuração do Neutron.

Segue-se agora para a explicação da estrutura interna do Neutron Server, incluindo o já mencionado *Core Plugin*. O Neutron Server é constituído na sua génese por vários componentes, estes estão representados na figura 5.2. A API RESTful é por onde toda a gestão das redes geridas pelo Neutron é feita, associado a esta, estão também as *API Extensions* que fornecem meios para estender a API com novas funcionalidades específicas, incluindo novas entidades. O *RPC Service*, que faz uso da API RPC do OpenStack o Oslo, tem como objetivo gerir a comunicação entre o Neutron Server e os outros componentes do mesmo, como por exemplo os *Plugin Agent*. O *Advanced Service Plugin* pretende fornecer uma API que permite a inclusão de serviços de rede externos na *Cloud* como por exemplo FaaS. O componente *Plugin* é o que terá mais importância, pois é através dele que são tratados os pedidos recebidos pela RESTful API e, de acordo com cada pedido o mesmo aplica as respetivas ações necessárias para configurar a rede conforme o pretendido. Este *Plugin* é chamado de *Core Plugin*, cada *Core Plugin* pode possuir um agente ou não, consoante o tipo de dispositivos que o

[2]https://www.mysql.com

[3]https://www.rabbitmq.com

mesmo vai configurar. Quando o *Core Plugin* é baseado em agentes, os agentes são tipicamente uma instância a ser executada, em cada máquina que possui diapositivos de rede virtuais que necessitem de ser configurados, para atender às necessidades de um determinado pedido. Cada *Core Plugin* está normalmente associado um mecanismo de rede L2 que permite a gestão de redes virtuais. Por exemplo existem *Core Plugins* para linuxbridge, OVS e Hyper-V e para controladores externos. Mais recentemente, estes *Core Plugins* monolíticos, que só permitem usar um mecanismo de rede L2 por *deployment*, foram substituídos por um *Core Plugin* que permite o uso de diferentes mecanismos de rede L2 na mesma rede virtual. Este *Core Plugin* permite também, que possam ser usados ao mesmo tempo agentes em máquinas com um tipo de tecnologia e em outras máquinas agentes com outro. Este novo *Plugin* tem o nome de ML2 *Plugin*, e vem tornar todos os *Core Plugins* mencionados anteriormente obsoletos devido as suas vantagens inerentes. Na secção seguinte é abordada com mais atenção o ML2, que vai ser onde a *framework* relativa a este trabalho vai ser integrada.

Como já foi mencionado o Neutron é um componente que foi construido para ser extensível, e por isso, proporciona várias formas de o fazer de acordo com as funcionalidades que se pretendem incluir. É possível, criar novas extensões à API RESTful por forma a receber pedidos de operações relacionadas com novas entidades ou funcionalidades. Estas extensões à API podem ser de três tipos diferentes, são elas extensões para introduzir novas entidades (tal como nas entidades base, é possível ter entidades específicas para implementar por exemplo, uma determinada operação num contexto específico de um *Core Plugin*). Extensões que permitem fazer uma operação específica num determinada recurso disponível. Por fim extensões que têm como objetivo adicionar novos atributos às entidades já existentes. Em que estas últimas são mais usadas em contextos específicos de uma ação relacionada com uma entidade num determinado *Core Plugin*.

O Neutron é capaz de isolar as redes virtuais dos seus utilizadores, entre os diversos nós do *deployment* OpenStack, usando túneis *overlay* (normalmente GRE ou VXLAN) ou usando VLANs. Quando o *deployment* do Neutron é baseado em VLANs, o gestor da infraestrutura de rede física deverá reservar um conjunto de VLAN IDs, de modo a poderem serem usados pelos utilizadores nas suas redes virtuais. O Neutron fará a alocação desses VLAN IDs quando necessitar através do uso da extensão *Provider Network*. É também possível fazer uso de um router físico para *gateway* das redes virtuais, embora estes não sejam tão flexíveis em relação ao uso de routers virtuais. Isto porque, com routers virtuais o utilizador tem a possibilidade de gerir os routers nas suas redes virtuais, podendo criar routers para cada uma delas de forma independente, para acesso a redes externas.

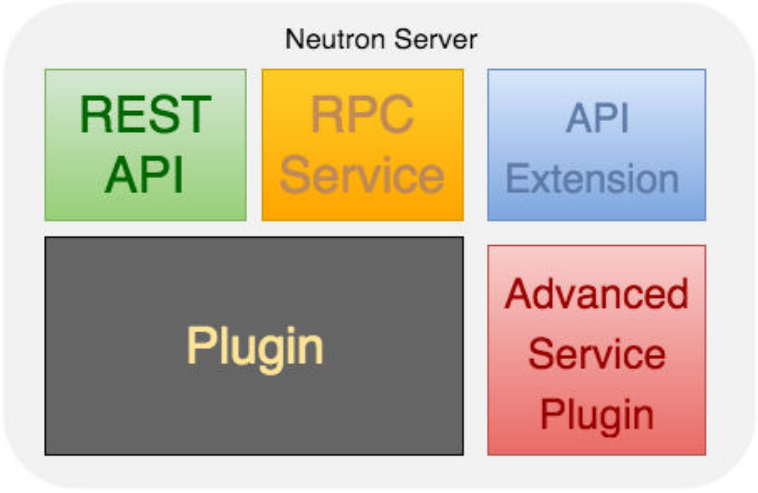

Figura 5.2: Estrutura interna do Neutron

5.1.3 NEUTRON ML2 CORE PLUGIN

O *Core Plugin* ML2 veio para substituir os antigos *Core Plugins* monolíticos. Este novo *Core Plugin* permite o uso de vários mecanismos de rede L2 em simultâneo na mesma instância do Neutron. Isto é conseguido através dos chamados *drivers*, *drivers* estes que podem ser carregados dinamicamente em *runtime*. Estes drivers são divididos em dois tipos, são eles os *Type Drivers* e os *Mechanism Drivers*. A arquitetura deste *Core Plugin* está representada na figura 5.3. Os *Type Drivers* têm como objetivo manter a informação necessária sobre o tipo de rede que suportam, validar o acesso a redes externas por parte de redes virtuais e fazer a reserva de redes virtuais, para os utilizadores de cada tipo de rede disponível. Os tipos de rede oficialmente suportados na versão *Mitaka* são *local*, *flat*, VLAN, GRE e VXLAN. Os *Mechanism Drivers* possuem como tarefa pegar na informação definida no *Type Driver*, e assegurar a aplicação das devidas configurações necessárias junto dos mecanismos disponíveis, para obter o efeito pretendido numa determinada rede virtual. Por serem estes os responsáveis pela aplicação das configurações pedidas pela API RESTful, estes devem implementar todas as operações relacionadas com as entidades base do Neutron (*Networks*, *Subnets* e *Ports*) e se suportarem alguma extensão da API, devem igualmente implementar todas as operações a ela relacionadas. Isto de forma a repercutir efetivamente, os pedidos de operações em ações específicas na rede, com vista a atingir o objetivo pretendido por parte dos utilizadores para a suas redes virtuais. Os *Mechanism Drivers* mais comuns e que atualmente fazem parte do ML2 são os que dão suporte para linuxbridge, Open vSwitch e Hyper-V. Existem também *Mechanism Drivers* que interagem com dispositivos físicos, são eles Cisco Nexus e o Arista.

Estes *Mechanism Drivers* podem possuir, da mesma forma que os *Core Plugins* monolíticos, agentes que comunicam com estes usando RPC que permitem interagir com os mecanismos existentes na máquina onde o mesmo está a correr. Ou no entanto, podem interagir diretamente com controladores, isto é, sem recurso a agentes por forma a aplicarem as configurações em dispositivos por eles controlados, residentes nos diferentes nós do *deployment* OpenStack. Os *Mechanism Drivers* disponíveis são declarados no ficheiro de configuração do Neutron, e quando é efetuado um pedido na API, estes são carregados em *runtime* em que a correspondente chamada, é feita em cada um deles de acordo com uma abordagem *top-down*. Dependendo do pedido, cada um vai ou não tomar ações que lhe competem por forma a

aplicar as alterações na rede para cumprir os objetivos do pedido.
Este *Core Plugin* tem vindo a substituir gradualmente os *Plugins* monolíticos, por oferecer a possibilidade de usar várias tecnologias de rede em simultâneo no mesmo *deployment* do Neutron. Outra razão prende-se também pelo facto, de ser facilmente expansível através das APIs disponibilizadas para desenvolvimento de novos *Type Drivers* e *Mechanism Drivers*. Tudo isto veio agilizar a adoção de novas tecnologias que permitem a virtualização de redes através dos *Type Drivers*, e a agilização da implementação das operações relacionadas com essas novas tecnologias através dos *Mechanism Drivers*.

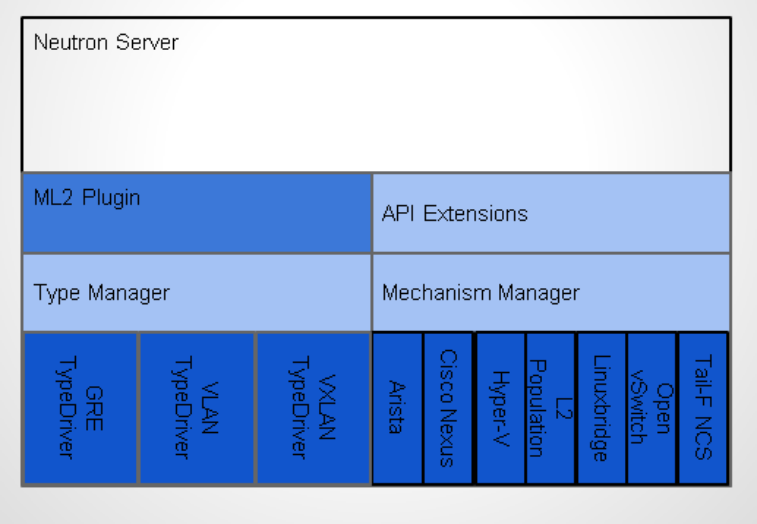

Figura 5.3: Estrutura interna do plugin ML2 [28]

5.1.4 Open vSwitch mechanism driver

É agora explorado mais em detalhe o Open vSwitch e o respetivo *Mechanism Driver* disponível no ML2 plugin, por este ser um elemento importante na solução encontrada associada a esta dissertação.

Open vSwitch

Open vSwitch[4] é uma implementação *open source* de um software que permite criar e gerir switches virtuais *multilayer*, que pretende fornecer uma solução de *switching* para ambientes de virtualização. Possui uma implementação que promove a automatização da rede através do uso de extensões criadas programaticamente, aliado a um bom suporte para um grande número de *standards* existentes atualmente na área de gestão de redes, tais como o OpenFlow e o OVSDB. O OVS é comparável ao robusto *linuxbridge* que é incluído por defeito nas distribuições Linux, trata-se de um módulo incorporado no kernel que permite para criar *bridges* para interligar as VM. O *Linuxbridge* de facto, fornece uma solução bastante eficiente e bem integrada, mas não tem o suporte para protocolos de gestão de rede como OVS possui, o que limita em muito a sua flexibilidade.

[4]http://openvswitch.org/

ML2 Open vSwitch mechanism driver

Este *Mechanism Driver* tem como objetivo configurar novas redes e portas nos switches virtuais a correr nas instâncias OVS nos diversos nós da *Cloud* OpenStack. É constituído pelo *Mechanism Driver* que recebe os pedidos, e pelos agentes que estão a correr em cada nó que possui uma instância OVS. Os agentes aplicam as configurações necessárias nos dispositivos do nó onde o agente está em execução, para fazer corresponder ao que foi pedido pela API do Neutron. Cada nó que necessite de serviços de rede, que numa instalação típica trata-se dos *Compute Nodes* e do *Controller Node*, possuem uma arquitetura de *bridges* que permite uma melhor gestão das redes virtuais das VMs e seu acesso ao exterior. A arquitetura de *bridges* usada num *Compute Node* em que a rede é gerida pelo *Mechanism Driver* OVS, está representada na figura 5.4, e é explicada a seguir:

- **br-int** - A *bridge* **br-int** é chamada de *bridge* de integração, a esta bridge são ligadas todas interfaces de rede virtuais das VMs criadas pelo pelo componente Nova. Os utilizadores que pretendem colocar na mesma rede L2 as suas VMs, necessitam de previamente criar uma nova rede, essa rede fica associada ao utilizador e pode ser usada para ligar no mesmo segmento L2 várias VMs pertencentes a este. Nesta bridge é feita a associação entre as VMs e uma determinada rede do utilizador dono das mesmas. Isso é feito, com recurso a VLANs em que os respetivos VLAN IDs são associados a cada rede virtual dos utilizadores. Cada porta de uma VM nesta bridge, possui os seu pacotes *tagged* com o respetivo VLAN ID correspondente à rede, à qual o utilizador pretende ligar a VM.
- **br-tun** - Esta *bridge* é usada, quando o tipo de *deployment* usado para interligar os nós que alojam os vários componentes, é baseado em túneis *overlay* (normalmente GRE e VXLAN). Neste tipo de *deployment*, o Neutron cria uma *mesh* de *links overlay* com uma tecnologia de *tunneling* pré-definida, em que cada nó tem uma ligação para cada um dos outros nós. Esta *bridge* tem como objetivo ser o *endpoint* de todos os túneis que interligam os vários nós existentes no *deployment*.
- **br-xxx** - A *bridge* **br-xxx** (em que "xxx" corresponde ao nome da interface que lhe está associada) é a *bridge* onde é ligada a interface física da máquina, que vai permitir que o tráfego das VMs nesse nó tenha acesso ao exterior.

As ligações que interligam estas *bridges* são as seguintes. As *bridges* **br-tun** e **br-xxx** são ligadas à *bridge* **br-int** por meio dos cabos virtuais de interligação de *bridges* chamados de *patch cables*. No caso de ser um *deployment* baseado em VLANs e para acesso às rede externas, o tráfego que fluí da *bridge* **br-int** para a **br-xxx** é substituído o VLAN ID interno associado à rede virtual por o VLAN ID na rede externa associado à rede virtual, no sentido inverso do tráfego a substituição dos IDs é feita de forma inversa. Quando o *deployment* é baseado em túneis, é substituído o VLAN ID interno pelo túnel ID associado à rede virtual na *mesh* de túneis.

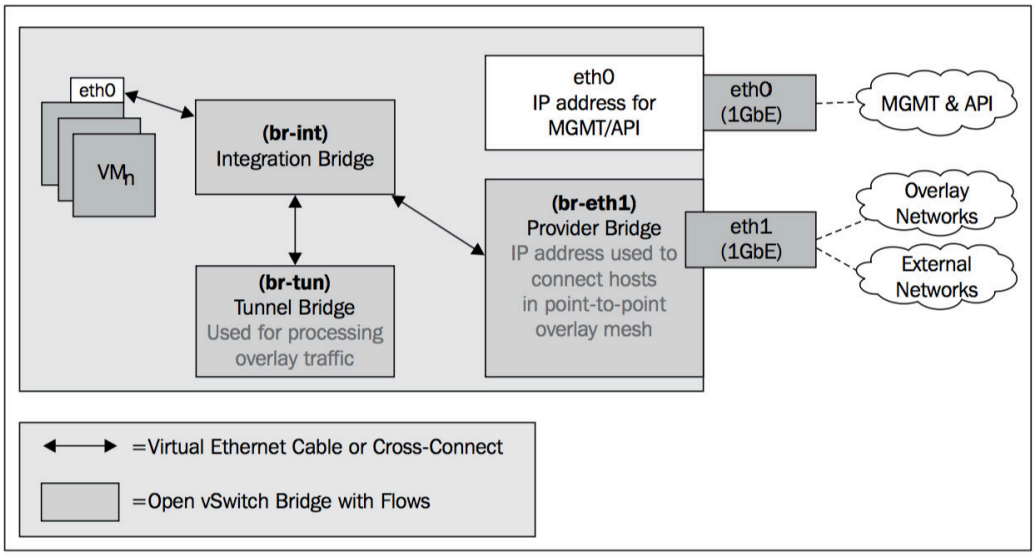

Figura 5.4: Arquitectura das *bridges* OVS num *Compute Node* [29]

5.2 EXTNET FRAMEWORK - INTRODUÇÃO

EXTNET foi o nome dado à *framework*, o nome EXTNET é originário das iniciais de EXTernal NETwork. Como já foi sendo mencionado ao longo deste documento, esta *framework* tem como objetivo estender as redes virtuais geridas por um CMS. Também já foi referido que a plataforma de *Cloud* escolhida para testar a *framework* foi o OpenStack que fornece os componentes necessários para criar uma *Cloud* IaaS, as razões desta escolha também já foram mencionadas anteriormente.
A *framework*, foi construida usando a linguagem Python, e é capaz de dotar um CMS dos mecanismos necessários para alcançar o objetivo, de estender as suas redes virtuais para a rede externa não gerida pelo mesmo. A *framework* é disponibilizada através de um *package* Python, que inclui a API base para dotar os CMS de funcionalidades de gestão de portas externas, e de seus respetivos links. Neste *package* são também incluídas APIs, para criação de mecanismos de recolha de informação sobre a topologia de rede externa, e para a construção de soluções para encontrar o melhor caminho, para a construção dos links relativos a uma porta externa recém criada.
A EXTNET *framework* é constituída por um conjunto de classes, que fazem um mapeamento quase direto dos módulos representados na figura 4.2. Estas classes vão ser incluídas no Neutron, tirando partido do paradigma de herança múltipla presente no Python, através de classes relativas a componentes chave na arquitetura do Neutron. Em relação às entidades apresentadas na secção 4.3.1, foi criado um modelo de base dados, que vai ser incorporado na base de dados principal do Neutron. Nesta implementação apenas foram usados dispositivos da marca Cisco, mais concretamente do tipo EtherSwitch com o sistema operativo IOS. Esta escolha deve-se ao facto, de existir a possibilidade de emular o seu sistema operativo em máquinas x86, ao facto de serem *managed* e de suportarem todos os tipos de virtualização de redes necessárias para efeitos de teste da *framework*. Estes suportam a criação de VLAN e túneis GRE. Estes switch, pelo facto de serem *managed* como já foi referido, possuem várias interfaces para se fazer a sua gestão, entre elas estão por exemplo telnet, SSH e SNMP. Desta forma, as operações de descoberta da topologia e as operações relacionadas com a criação de

paths para as portas externas, são implementados de forma a suportar estes dispositivos.

5.3 EXTNET FRAMEWORK - GESTÃO E PERSISTÊNCIA DAS ENTIDADES

5.3.1 PERSISTÊNCIA DAS ENTIDADES

Em relação à persistência da informação necessária ao funcionamento da *framework*, seguiu-se a opção de optar pelo uso das faculdades existentes no OpenStack já que este possui uma API bem estruturada e construida de forma a ser facilmente expandida. Possui ainda outras capacidades importantes do contexto da persistência da informação como por exemplo, um mecanismo de upgrade às entidades no futuro, sem comprometer a perda da informação já existente. Com isto a *framework* torna-se mais simples, por não estar a implementar uma funcionalidade que está disponível, e que se normalmente se pode tirar partido nos CMS.
Partindo das entidades mencionadas na secção 4.3.1, foram definidas as relações entre elas que estão representadas no seguinte diagrama relativo à base de dados representado na figura 5.5.

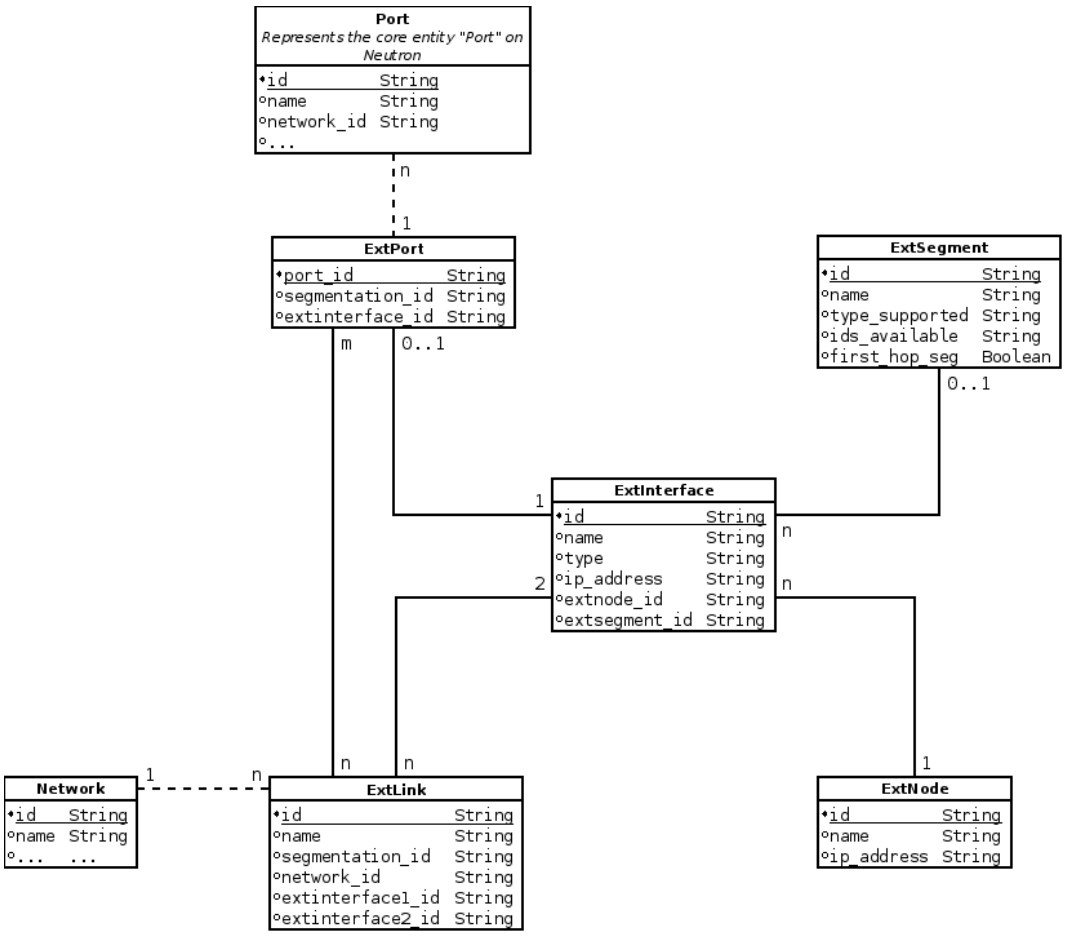

Figura 5.5: Diagrama da base dados da *framework* EXTNET

Durante o inicio da implementação, houve a perceção de que era necessário fazer de algumas alterações nas entidades, de modo a definir o modelo de base de dados final. Uma delas é em relação à *ExtPort*, que vai representar uma porta física ou virtual associada a uma interface de um dispositivo que se pretende que fique na mesma rede virtual do *datacenter*. Isto levou a que fosse necessário introduzir uma nova entidade que representasse diretamente as interfaces nos dispositivos de rede (estas interfaces podem tomar forma em vários tipos, por exemplo uma interface RJ45, um SSID, etc.). Esta nova entidade, pode albergar várias portas virtuais *ExtPort*, que estão associadas a uma interface de cada dispositivo que se pretende colocar na rede virtual. Podem também, servirem como *endpoints* para os *links* virtuais (*ExtLinks*) a serem criados para conduzir o tráfego até às *ExtPorts*. Esta nova entidade tem o nome de *ExtInterface*. As *ExtPort* vão ser uma extensão de uma entidade *core* já existente no Neutron chamada de *Port*. Fazendo uso desta entidade como base, é possível tirar partido de todos os mecanismos incluídos no Neutron para gestão da rede IP, principalmente o DHCP. Como já foi mencionado, uma *ExtPort* é uma extensão da entidade *Port* e está associada diretamente a uma *ExtInterface*. Uma *ExtInterface* representa uma interface existente num dispositivo de rede, as *ExtInterfaces* podem ter associada uma ou várias *ExtPort* ou um *ExtLink*, nunca dos diferentes tipos ao mesmo tempo. É possível então observar que, uma *ExtPort* pode ter vários *ExtLink* que constituem a ligação da *Cloud* OpenStack até a localização na rede externa dessa *ExtPort* associada a uma *ExtInterface*, formando assim um *path* para uma rede virtual na rede externa. Um *ExtLink* pode estar associado a várias *ExtPorts* que fazem uso deste para, na mesma rede virtual, conduzirem o tráfego por um segmento (*ExtSegment*) da rede física. Os *ExtLinks* estão associados a uma rede virtual no OpenStack e possuem como *endpoints* duas *ExtInterfaces*.
No caso dos *ExtSegments*, estes não vão sofrer alterações em relação à funcionalidade apresentada na arquitetura. Os *ExtSegments* podem conter várias *ExtInterfaces* que servirão, de *endpoints* para criar *ExtLink* associados a redes virtuais que passam por esse segmento. De igual forma, os *ExtNodes* a sua definição e características não são alteradas, em relação ao que foi apresentado anteriormente na arquitetura. Tal como o *ExtSegment*, os *ExtNodes* são constituídos por um conjunto de *ExtInterfaces*. As relações entre as entidades podem ser consultadas no diagrama de classes da figura 5.6.

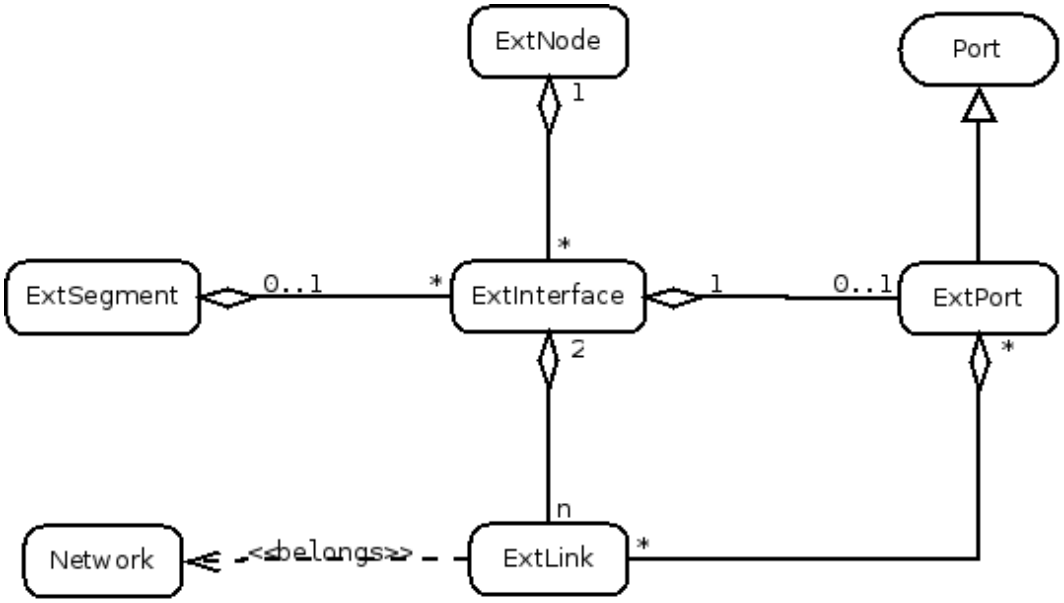

Figura 5.6: Diagrama de classes das entidades da *framework* EXTNET

É descrito agora o significado de cada atributo das entidades apresentadas na figura 5.5 nas seguintes tabelas.

Atributo	**Tipo**	**Descrição**
port_id	String-uuid	Representa o ID de uma porta externa. É chave primária e chave estrangeira da tabela relativa à entidade *core* Port.
segmentation_id	String	Atributo que contém o ID de segmentação, relativo à rede virtual onde se pretende ligar a porta externa.
extinterface_id	String	Chave estrangeira relativa à tabela ExtInterface, que faz a relação que diz que uma *ExtPort* está associada a uma *ExtInterface*.

Tabela 5.1: Descrição dos atributos relativos a uma `ExtPort`

Atributo	**Tipo**	**Descrição**
id	String-uuid	Representa o ID de um *ExtLink* e é chave primária desta entidade.
name	String	Nome *human-readable* do *ExtLink*.
segmentation_id	String	Contém o ID de segmentação do *ExtLink*, este ID pode ser o Túnel ID ou o VLAN ID. Consoante o tipo de tecnologia utilizada, ou seja um túnel *overlay* ou VLAN.
network_id	String-uuid	Contém o network ID da rede virtual associada ao *ExtLink*.
extinterface1_id	String	Chave estrangeira relativa à tabela ExtInterface referente ao primeiro *endpoint* do *ExtLink*.
extinterface2_id	String	Chave estrangeira relativa à tabela ExtInterface referente ao segundo *endpoint* do *ExtLink*.

Tabela 5.2: Descrição dos atributos relativos a um `ExtLink`

Atributo	Tipo	Descrição
id	String-uuid	Representa o ID de um *ExtInterface* e é chave primária desta entidade.
name	String	Nome *human-readable* do *ExtInterface*.
type	String	Contém informação sobre o tipo da *ExtInterface*, ou seja, se é uma porta física L2 ou L3, um SSID de uma rede Wi-Fi ou outra realização equivalente. Nesta implementação os valores possíveis para este atributo são L2 ou L3.
ip_address	String	Endereço IP associado à interface que esta *ExtInterface* representa.
extnode_id	String-uuid	Chave estrangeira que representa a relação entre uma *ExtInterface* e um *ExtNode*. Em que um *ExtNode* pode conter várias *ExtInterface*.
extsegment_id	String	Chave estrangeira representativa da relação entre uma *ExtInterface* e um *ExtSegment*. Em que uma *ExtInterface* pertence a um *ExtSegment*.

Tabela 5.3: Descrição dos atributos relativos uma `ExtInterface`

Atributo	Tipo	Descrição
id	String-uuid	Representa o ID de um *ExtSegment* e é chave primária desta entidade.
name	String	Nome *human-readable* do *ExtSegment*.
type_supported	String	Contém informação acerca do tipo da tecnologia que ambas as *ExtInterfaces* associadas a este suportam. Este atributo nesta implementação pode conter valores as *strings* VLAN ou GRE.
ip_address	String	Endereço IP associado à interface que esta *ExtInterface* representa.
extnode_id	String-uuid	Chave estrangeira que representa a relação entre uma *ExtInterface* e um *ExtNode*. Em que um *ExtNode* pode conter várias *ExtInterface*.
extsegment_id	String	Chave estrangeira representativa da relação entre uma *ExtInterface* e um *ExtSegment*. Em que uma *ExtInterface* pertence a um *ExtSegment*.

Tabela 5.4: Descrição dos atributos relativos um `ExtSegment`

Atributo	Tipo	Descrição
id	String-uuid	Representa o ID de um *ExtNode* e é chave primária desta entidade.
name	String	Nome *human-readable* do *ExtNode*.
ip_address	String	Endereço IP associado à interface usada para administração disponível no *ExtNode*.

Tabela 5.5: Descrição dos atributos relativos um `ExtNode`

5.3.2 GESTÃO DO CICLO DE VIDA DAS ENTIDADES

Com esta *framework*, pretende-se que se torne possível a criação de portas fora da rede gerida pelo OpenStack para as redes virtuais, isto com o mínimo de comandos ou operações na *Graphical User Interface* (GUI) possível. É possível a criação de portas externas usando um só comando, em que a seguir a *framework* trata de pesquisar pela topologia da rede externa caso seja necessário, criando um mapeamento da mesma na base de dados. Através de um algoritmo de mapeamento pré-definido a *framework* é capaz de criar *paths* constituídos por links *overlay* criados segmento a segmento. Isto de forma a isolar o tráfego da rede gerida pelo OpenStack até à nova porta criada. Tudo isto envolve a criação de várias novas entidades na base de dados, torna-se por vezes necessário para pequenos ajustes, ter algo mais que ajude a gerir essas entidades geradas pela pela descoberta da topologia e

pelo mapeamento de rede.

Existe por vezes a impossibilidade de alguns dos dispositivos da rede externa não poderem ser descobertos por uma possível implementação de um mecanismo de descoberta da topologia, devido a incompatibilidades com o as *Management Information Base* (MIB) dos dispositivos de rede por exemplo. Por todas essas razões optou-se também por desenvolver extensões à API RESTful do Neutron que vão permitir uma gestão manual das entidades, podendo através destas criar novas e alterar as já existentes. Esta API RESTful serve de ponto de recolha de dados para as interfaces de interação do utilizador com o Neutron. As interfaces fornecidas por defeito no Neutron são o Horizon (projeto que pretende dotar o OpenStack de uma web dashboard) e uma CLI (projeto com o nome de `python-neutronclient`). Foram desenvolvidas as extensões correspondentes para a interface CLI `python-neutronclient` para interagir com as novas entidades presentes na API. As extensões desenvolvidas para a API estão descritas no apêndice E. Nas tabelas seguintes são apresentados os comandos disponibilizados na CLI.

Comando	Argumentos	Descrição	Exemplo
`extnode-create`	`EXTNODE_NAME --ip-address IP_ADDRESS`	Cria um novo `ExtNode`	`neutron extnode-create node1 --ip-address 192.168.2.1`
`extnode-update`	`EXTNODE_NAME --ip-address IP_ADDRESS`	Faz update à informação relativa a um `ExtNode`	`neutron extnode-update b39fbea6-2f58-4f52-a58d-bc9b5aca00a6 --ip-address 192.168.2.1`
`extnode-delete`	`EXTNODE_ID`	Apaga um `ExtNode`	`neutron extnode-delete b39fbea6-2f58-4f52-a58d-bc9b5aca00a6`
`extnode-list`		Faz a listagem de todos os `ExtNode`	`neutron extnode-list`
`extnode-show`	`EXTNODE_ID`	Mostra a informação relativa ao `ExtNode` com o ID fornecido	`neutron extnode-show b39fbea6-2f58-4f52-a58d-bc9b5aca00a6`

Tabela 5.6: Comandos da CLI relativos ao `ExtNode`

Comando	Argumentos	Descrição	Exemplo
`extsegment-create`	`EXTSEGMENT_NAME --type-supported TYPE_SUPPORTED --ids-available IDS_AVAILABLE`	Cria um novo `ExtSegment`	`neutron extsegment-create seg1 --type-supported vlan --ids-available 1000:2000`
`extsegment-update`	`EXTSEGMENT_ID --type-supported TYPE_SUPPORTED --ids-available IDS_AVAILABLE`	Faz update à informação relativa a um `ExtSegment`	`neutron extsegment-update b39fbea6-2f58-4f52-a58d-bc9b5aca00a6 seg1 --type-supported vlan --ids-availalbe 1000:2000`
`extsegment-delete`	`EXTSEGMENT_ID`	Apaga um `ExtSegment`	`neutron extsegment-delete b39fbea6-2f58-4f52-a58d-bc9b5aca00a6`
`extsegment-list`		Faz a listagem de todos os `ExtSegment`	`neutron extsegment-list`
`extsegment-show`	`EXTSEGMENT_ID`	Mostra a informação relativa ao `ExtSegment` com o ID fornecido	`neutron extsegment-show b39fbea6-2f58-4f52-a58d-bc9b5aca00a6`

Tabela 5.7: Comandos da CLI relativos ao `ExtSegment`

Comando	Argumentos	Descrição	Exemplo
extinterface-create	EXTINTERFACE_NAME --extnode-id EXTNODE_ID --ip-address IP_ADDRESS --extsegment-id EXTSEGMENT_ID	Cria um novo ExtInterface	neutron extinterface-create f0/1 --extnode-id b39fbea6-2f58-4f52-a58d-bc9b5aca00a6 --ip-address 192.168.0.2 --extsegment-id b39fbea6-2f58-4f52-a58d-bc9b5aca00a5
extinterface-update	EXTINTERFACE_ID --extnode-id EXTNODE_ID --ip-address IP_ADDRESS --extsegment-id EXTSEGMENT_ID	Faz update à informação relativa a uma ExtInterface	neutron extinterface-update b39fbea6-2f58-4f52-a58d-bc9b5aca00a6 --ip-address 192.168.0.2 --extsegment-id b39fbea6-2f58-4f52-a58d-bc9b5aca00a5
extinterface-delete	EXTINTERFACE_ID	Apaga um ExtInterface	neutron extinterface-delete b39fbea6-2f58-4f52-a58d-bc9b5aca00a6
extinterface-list		Faz a listagem de todos os ExtInterface	neutron extinterface-list
extinterface-show	EXTINTERFACE_ID	Mostra a informação relativa ao ExtInterface com o ID fornecido	neutron extinterface-show b39fbea6-2f58-4f52-a58d-bc9b5aca00a6

Tabela 5.8: Comandos da CLI relativos à ExtInterface

Comando	Argumentos	Descrição	Exemplo
extlink-create	EXTLINK_NAME --extinterface1-id EXTINTERFACE1_ID --extinterface2-id EXTINTERFACE2_ID --network-id NETWORK_ID	Cria um novo ExtLink	neutron extlink-create link1 --extinterface1-id 469c7b32-6836-4e43-a7fc-0de78f689371 --extinterface2-id d8cde7bd-b066-4c2e-8ac1-9506542935a0 --network-id 4e9acc9d-869a-4f04-a617-d9670f32526b
extlink-update	EXTLINK_ID EXTLINK_NAME	Faz update à informação relativa a um ExtLink	neutron extlink-update b39fbea6-2f58-4f52-a58d-bc9b5aca00a6 link2
extlink-delete	EXTLINK_ID	Apaga um ExtLink	neutron extlink-delete b39fbea6-2f58-4f52-a58d-bc9b5aca00a6
extlink-list		Faz a listagem de todos os ExtLink	neutron extlink-list
extlink-show	EXTLINK_ID	Mostra a informação relativa ao ExtLink com o ID fornecido	neutron extlink-show b39fbea6-2f58-4f52-a58d-bc9b5aca00a6

Tabela 5.9: Comandos da CLI relativos ao ExtLink

A tabela seguinte representa uma *ExtPort* que neste caso vai ser uma extensão à entidade Port já existente no Neutron. Esta extensão pretende adicionar os atributos necessários à entidade Port transformando-a numa ExtPort, podendo assim tirar partido de alguns dos serviços relacionados com as portas do Neutron (como por exemplo o DHCP). Apenas vão ser mencionados na tabela os novos atributos relevantes relacionados com as operações relativas à porta externa. Importa também referir o facto de que, deve ser fornecido à entidade Port o endereço MAC do dispositivo à qual a porta vai ficar associada, isto vai permitir que o servidor de DHCP que está a correr no Neutron possa fornecer-lhe um endereço IP da gama de endereços atribuídos à rede virtual em questão.

Comando	Argumentos	Descrição	Exemplo
port-create	NETWORK_ID --mac-address MAC_ADDRESS --extinterface-name EXTINTERFACE_NAME --extnode-name EXTNODE_NAME	Cria um novo ExtNode	neutron port-create b39fbea6-2f58-4f52-a58d-bc9b5aca00a6 --mac-address 00:11:22:33:44:55 --extinterface-name f1/0 --extnode-name node1
port-delete	PORT_ID	Apaga uma Port	neutron port-delete b39fbea6-2f58-4f52-a58d-bc9b5aca00a6
port-list		Faz a listagem de todos os Port	neutron port-list
port-show	PORT_ID	Mostra a informação relativa ao Port com o ID fornecido	neutron port-show b39fbea6-2f58-4f52-a58d-bc9b5aca00a6

Tabela 5.10: Comandos da CLI relativos à ExtPort

5.4 EXTNET FRAMEWORK - NEUTRON - INTEGRAÇÃO DOS MÓDULOS

Nesta secção são abordadas as formas encontradas para integrar a arquitetura definida na secção 4.3.2 no CMS OpenStack. Cada módulo irá ser integrado nos respetivos componentes que fazem parte do funcionamento interno do Neutron, usando o paradigma de herança múltipla. Será explicado módulo a módulo, de que forma e em que componente foi adicionado, e outras considerações tomadas aquando desta inclusão dos módulos da *framework* nos componentes do Neutron. Na figura 5.7 está representado o diagrama de componentes da *framework* e respetiva integração no Neutron.

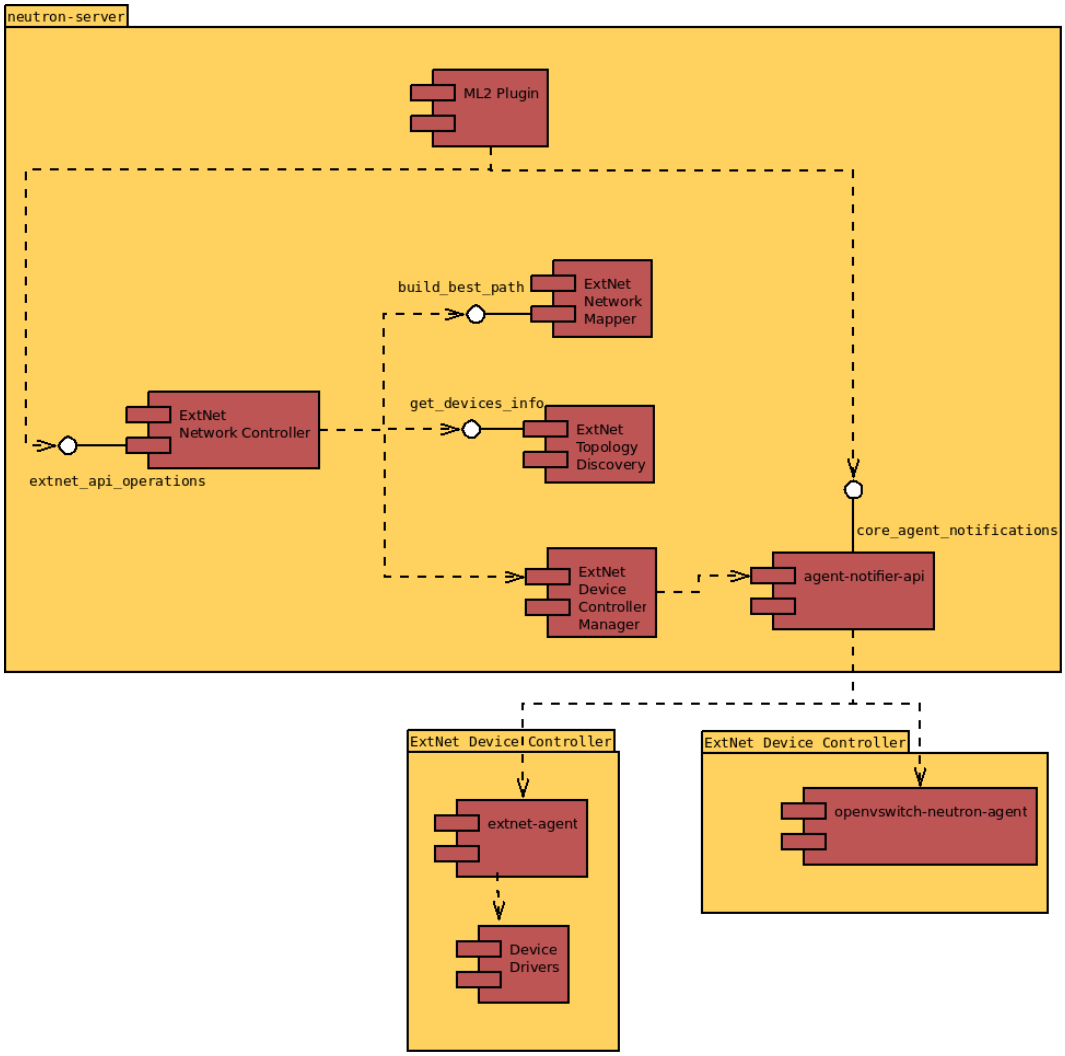

Figura 5.7: Diagrama de componentes da *framework* EXTNET integrados no Neutron

5.4.1 NETWORK CONTROLLER

Embora a escolha natural para integrar o *Network Controller* no Neutron fosse através de um *Mechanism Driver*, por o *Network Controller* encaixar de forma quase perfeita no conceito do mesmo, não se optou por essa abordagem. Isto devido ao facto de que, seria necessário fazer alterações profundas ao nível da gestão dos *Mechanism Drivers*, ao ter a necessidade de dotar estes de capacidades para responder a pedidos relacionados com as novas entidades adicionadas ao Neutron. Como consequência iria levar a que fossem necessárias bastantes alterações à implementação de gestão dos *Mechanism Drivers* no Neutron, e logo iria contra a ideia que a *framework* é que tem de se adaptar ao CMS e não o contrário. Tendo essa consideração em mente, optou-se então por integrar o *Network Controller* no Neutron como uma classe base do *Plugin* ML2. Com isto foi possível adicionar ao ML2, as capacidades necessárias para o mesmo controlar a rede externa, permitindo-nos tirar todo o partido das funcionalidades existentes em relação à base de dados, das capacidades existentes de comunicação com outros componentes usando o mecanismo de mensagens RPC e da API RESTful.
Tem a seu cuidado toda a lógica da gestão dos *paths* das portas externas, a receção dos pedidos

relativos às extensões adicionadas à API RESTful, a execução de operações na rede se necessário e a salvaguarda das informações relativas às entidades na base de dados. É no *Network Controller* que são instanciados os componentes *Topology Discovery* para a descoberta da topologia da rede externa, o *Network Mapper* que permite a fazer o mapeamento dos *paths* para as portas externas de acordo com o algoritmo previamente definido, e o *Device Controller Manager* que faz a gestão da comunicação com os diversos *Device Controllers*. Possui a implementação dos métodos `create_extport`, `delete_extport`, `create_extlink` e `delete_extlink` que são métodos que para além de fazerem operações na base de dados, necessitam de fazer alterações na rede física externa. Os restantes métodos apenas fazem operações ao nível da base de dados. O método `create_extport` pode necessitar de fazer uma busca na topologia, se não encontrar o dispositivo na base de dados, criar um *path* formado por `ExtLinks` até ao dispositivo onde se encontra a interface onde vai ser colocada a *ExtPort*, se esta ainda não tiver nenhuma associada. Se já houver uma *ExtPort* na mesma rede virtual associada nessa *ExtInterface*, a mesma é criada fazendo uso do *path* já existente para a sua congénere. No caso do método *delete_extport*, este vai apagar a referência dessa *ExtPort* da *ExtInterface* que lhe estava associada e verificar *ExtLink* a *ExtLink*, daqueles faziam parte do *path*, os que possuem mais alguma `ExtPort` associada. Os que ainda tiverem `ExtPort` associadas são mantidos, os que não tiverem nenhuma são automaticamente desconfigurados dos dispositivos e eliminados. Com isto consegue-se ter na rede configurados apenas os `ExtLink` que estão efetivamente a ser usados por uma ou mais `ExtPort`. Existem também outros métodos auxiliares, que são usados no apoio às funcionalidades apresentadas nos métodos mencionados anteriormente.
Os métodos que o *Network Controller* implementa, estão disponíveis para consulta no apêndice D na *listing* 6.

5.4.2 TOPOLOGY DISCOVERY

O componente *Topology Discovery* tem como objetivo fornecer um meio capaz de percorrer a rede externa, de forma a recolher a informação relevante para o contexto da *framework*. Trata-se de uma classe com o nome de `TopologyDiscovery` que possui dois métodos. O `__init__` que recebe como parâmetros um objeto, que contém o mecanismo de acesso aos dispositivos (este objeto pode ter por exemplo uma implementação baseada em SNMP, SSH ou telnet), neste caso o mecanismo escolhido foi o SNMP orientado para recolher informação das MIBs Cisco. Pode no entanto ser configurado para outras marcas e modelos de dispositivos, mas para efeitos deste trabalho apenas foram considerados os dispositivos Cisco. Portanto esta implementação do *Topology Discovery* apenas recolhe informação de dispositivos Cisco. O outro método presente na classe `TopologyDiscovery` tem o nome de `get_devices_info`, este é um método recursivo que percorre dispositivo a dispositivo e recolhe as informações necessárias dos dispositivos, para a construção da topologia da rede externa, para posterior uso como base de informação para as operações no contexto da *framework*. Este método faz uso do objeto que contém o mecanismo de acesso às MIB dos dispositivos, que é passado como argumento aquando da instanciação do `TopologyDiscovery`, isto para saber como aceder à informação pretendida dos dispositivos.
O seu método de funcionamento está representado na figura 5.8 e é explicado a seguir. O módulo `TopologyDiscovery` liga-se a um dispositivo, recolhe toda a informação necessária desse dispositivo e verifica através dessa informação se existe dispositivos vizinhos ao qual é possível ligar-se. Se existirem vai se ligar a cada um deles e repete o ciclo. Antes de fazer a ligação a um dispositivo, verifica através

do endereço IP e do *hostname* se esse dispositivo já foi visitado anteriormente, se já o tiver feito não é novamente visitado. No fim quando já todos foram visitados é retornado um dicionário com toda a informação relativa aos dispositivos. É através dessa informação acerca da topologia da rede externa que vai permitir definir *paths* para as portas externas. A implementação da função `get_devices_info` é apresentada no apêndice C, as assinaturas dos restantes métodos estão disponíveis no apêndice D na *listing* 7.

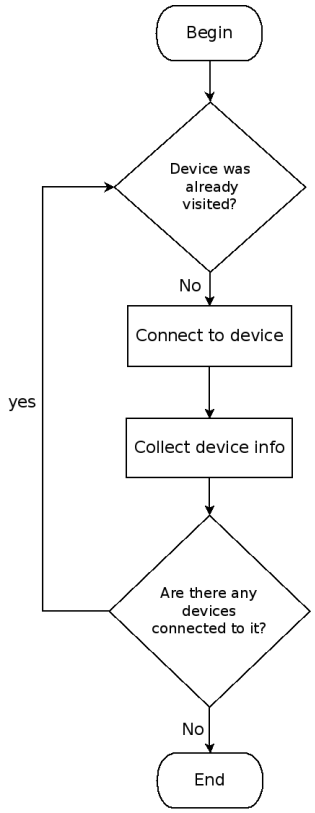

Figura 5.8: Diagrama de fluxo representativo do funcionamento do módulo *Topology Discovery*

5.4.3 NETWORK MAPPER

O *Network Mapper* tem como principal objetivo fornecer a assinatura para uma função, que permite definir de que forma o caminho para as portas externas é definido. Por outras palavras, fornece uma interface para a implementação do algoritmo de pesquisa, de forma a descobrir o melhor caminho aquando da criação de uma porta externa. Nesta implementação podem ser ainda adicionados parâmetros específicos que podem influenciar o comportamento do algoritmo implementado. Para este trabalho foi usado um algoritmo de pesquisa do melhor caminho, cuja a implementação base em Python usada está disponível no site da linguagem [30], trata-se de uma implementação otimizada para Python com um grande grau de simplicidade. O *Network Mapper* é implementado na classe `ExtNetNetworkMapperSP`, e contém um método de nome `build_best_path` onde é fornecido o grafo representativo da topologia de rede externa. O ponto de inicio é o nó onde está alojado o Neutron (neste caso o chamado *Controller Node*), e o ponto final será o dispositivo da rede externa que irá

disponibilizar a porta externa (*ExtPort*). Este método retorna uma lista com os dispositivos, por onde deverão ser criados links (*ExtLinks*) que formarão o *path*, até à ao dispositivo que vai albergar a porta externa (*ExtPort*). O métodos necessários a implementar estão disponíveis no apêndice D na *listing* 8.

5.4.4 DEVICE CONTROLLER MANAGER

O *Device Controller Manager* tem como tarefa gerir as ligações aos *Device Controllers*, e distribuir os pedidos de configuração provenientes do *Network Controller*. No âmbito desta implementação vão existir dois, e ambos usam o mecanismo RPC mais concretamente o `q-agent-notifier` que tem como fim notificar os agentes de mudanças a serem efetuadas na rede. Um dos *Device Controller* vai ser o já existente *neutron-openvswitch-agent* ao qual foram adicionados meios para comunicar com o *Device Controller Manager*. Este agente no contexto da *framework*, está encarregue de configurar o switch virtual OVS presente no *Controller Node*, quando as redes virtuais criadas pelo Neutron são baseadas em túneis. Quando as redes virtuais são baseadas em VLAN não é necessário criar o túnel, visto que no *first-hop* o Neutron faz uso dos VLAN IDs a ele disponibilizados para atribuir às redes virtuais. A *framework*, no caso das VLAN verifica qual o VLAN ID associado à respetiva VLAN e faz uso dessa associação, para criar um *ExtLink* que a representa perante a *framework*. Voltando ao caso de ser um *deployment* baseado em túneis, ao *neutron-openvswitch-agent* foram adicionados mecanismos para criar túneis GRE até ao *gateway* do *datacenter*, daí o *neutron-openvswitch-agent* ser também um *Device Controller*. Este *endpoint* criado pelo *neutron-openvswitch-agent* é colocado na `br-int`, fazendo o mapeamento da VLAN ID interna com o túnel ID do túnel criado pelo agente. A partir do *gateway* a gestão das configurações é feita por outro *Device Controller* a que lhe foi dado o nome de `extnet-agent`. O `extnet-agent` recebe as configurações do *Device Controller Manager* através de RPC pelo `q-agent-notifier`, vai verificar a que dispositivo se destinam, carrega o *Device driver* correspondente ao dispositivo em *runtime* e aplica as configurações de forma remota. Este *Device Controller Manager* é um componente passado como um objeto ao *Network Controller*. Os métodos necessários para o *Device Controller Manager* encontram-se na *listing* 9 para efeitos de consulta. As chamadas a métodos do agente EXTNET do lado do *Device Controller Manager*, são efetuadas fazendo chamadas a métodos identificados por tópicos, em que estes tópicos são registados por consumidores que pretendem receber as mensagens. O agente EXTNET regista um conjunto de tópicos tornando-se assim num consumidor, podendo assim, receber chamadas ao seus métodos por parte do *Device Controller Manager*. Uma chamada a um dos métodos do agente EXTNET é exemplificada no trecho de código 1.

```
topic = self.get_device_controller(dev_ctrl_name)
topic_create_extport = topics.get_topic_name(topic,
                                             topics.EXTNET_PORT,
                                             topics.CREATE)
  target = oslo_messaging.Target(topic=topic, version='1.0')
  client = n_rpc.get_client(target)
  cctxt = client.prepare(topic=topic_create_extport,
                         fanout=False,
                         timeout=30)

  return cctxt.call(context,
```

```
        'deploy_port',
        segmentation_id=segmentation_id,
        node=node,
        interface=interface)
```

Listing 1: Trecho de código relativo a uma chamada a um método do agente EXTNET por parte do Device Controller Manager.

5.4.5 DEVICE CONTROLLER

Os *Device Controller* tem a responsabilidade de ter ao seu cuidado um conjunto de dispositivos (os dispositivos que são atribuídos a um *Device Controller* são descritos no ficheiro de configuração do Neutron), em que quando pedido pelo `Device Controller Manager` estes aplicam as configurações que lhe são enviadas. Por outras palavras, é um componente que conhece a forma de como comunicar com o dispositivo (através dos *Device Drivers*) que, ao receber um pedido de configuração, consegue ligar-se a este e aplicar as configurações necessárias. Isto para os dispositivos que lhe foram previamente atribuídos. Quando lhe chega um pedido de configuração, o *Device Controller* tem a capacidade de carregar o respetivo `Device Driver` em *runtime* e aplicar as configurações pedidas. Como já foi abordado na secção 5.4.4 esta implementação é constituída por dois *Device Controller* e ambos usam como meio de comunicação a implementação de um mecanismo RPC pelo OpenStack chamado de *oslo-rpc*[5] para comunicar com o *Device Controller Manager*. As assinaturas dos métodos estão disponíveis na *listing* 10.

EXTNET AGENT

O *Device Controller* dos dispositivos externos físicos, como já foi mencionado, a sua implementação foi baseada num agente do Neutron. Ao usar um agente é possível fazer uso de todas as capacidades que estes oferecem, tanto a nível do *lifecycle* do processo que é lançado, pela facilidade com que se pode fazer uso do serviço de mensagens *oslo-rpc* para comunicar com o `neutron-server`. Este agente é lançado quando o Neutron arranca, e fica a aguardar por pedidos de configuração provenientes do *Device Controller Manager* destinados aos dispositivos externos. Relativamente à comunicação entre o agente e o *Device Controller Manager*, no trecho de código 2, é possível consultar de que forma esta é feita. O trecho de código representa o *deploy* de um conjunto de consumidores para as chamadas a métodos do agente. Cada consumidor é constituído, neste caso, por um par de tópicos. Em que cada par de tópicos, fica à espera de receber chamadas a métodos associados a esse par. Por exemplo, o consumidor `[topics.EXTNET_PORT, topics.CREATE]` recebe chamadas para o método `deploy_port`. Os restantes métodos, são chamados ao receber chamadas nos tópicos correspondentes de uma forma análoga à do exemplo anterior.

```
# RPC network init
  self.context = context.get_admin_context_without_session()
```

[5]https://wiki.openstack.org/wiki/Oslo/Messaging

```
# Define the consumers for the agent
consumers = [[topics.EXTNET_PORT, topics.CREATE],
             [topics.EXTNET_LINK, topics.CREATE],
             [topics.EXTNET_PORT, topics.DELETE],
             [topics.EXTNET_LINK, topics.DELETE], ]
self.connection = agent_rpc.create_consumers([self],
                                             topics.EXTNET_AGENT,
                                             consumers,
                                             start_listening=True)
```

Listing 2: Trecho de código que permite criar um conjunto de consumidores de mensagens para o EXTNET Agent.

5.4.6 DEVICE DRIVER

Os *Device Driver* possuem as capacidades necessárias para se ligarem aos dispositivos e aplicarem as configurações para obter o comportamento desejado pela *framework*. Cada *Device Driver* é construido para interagir com um determinado tipo de dispositivo (pode ser por marca, modelo ou tipo dependendo do grau de normalização). A comunicação pode ser feita através de um qualquer tipo de ligação desde que o dispositivo a suporte, e que esteja de acordo com os princípios de segurança pré-definidos (pode ser telnet, SSH, SNMP, ou outro do género). Os *Device Driver* como já foi mencionado também na secção 5.4.4, são instanciados em *runtime* aquando da chegada de um pedido de configuração. Este pedido tem de ser destinado a um dispositivo, que tenha sido configurado previamente nas definições com um determinado *Device Driver*. Para fazer a ligação aos dispositivos de rede externa, foi usado o *package* `pexpect` que permite ligar ao dispositivo usando as mais conhecidas formas de gestão remota (telnet, SSH, entre outros), depois da ligação feita, são executados os comandos ou *scripts* usando a linguagem Python tal qual como se fosse um humano a fazer a sua inserção. Cada *Device Driver* vai estar associado a um dispositivo especifico (por exemplo a um dispositivo de uma determinada marca e de um determinado modelo), e os ficheiros de configuração, são destinados a cada dispositivo individual com um *hostname* diferente entre eles. Estes ficheiros de configuração são colocados na mesma pasta dos *Device Drivers*, estão escritos no formato *JavaScript Object Notation* (JSON) e possuem informação relativa ao *setup* da ligação para aplicação das configurações e informação relativa às configurações (por exemplo na implementação deste trabalho, é guardado algum contexto relativo às configurações realizadas nos dispositivos Cisco. Este facto é abordado na secção 5.6).
O protocolo usado para testes na implementação, para aplicar as configurações nos dispositivos Cisco foi o Telnet. As assinaturas dos métodos relativos aos *Device Drivers* estão disponíveis na *listing* 11

5.5 EXTNET FRAMEWORK - NEUTRON - COMUNICAÇÃO ENTRE OS MÓDULOS DA *framework* DURANTE A CONSTRUÇÃO DE *paths* NA REDE EXTERNA

Os *paths* na rede externa, como já foi referido, são constituídos por vários *ExtLinks*. Para a construção dos *ExtLinks*, são precisas serem feitas várias trocas de mensagens entre os módulos da *framework*. Na figura 5.9, é possível ver de que forma se processam as trocas de mensagens entre os módulos da *framework* durante o *deploy* de um *ExtLink*. Neste caso, temos na figura um *setup* composto por um *Device Controller* e dois dos seus *Device Drivers*. Cada *Device Driver* está associado a um dispositivo distinto.

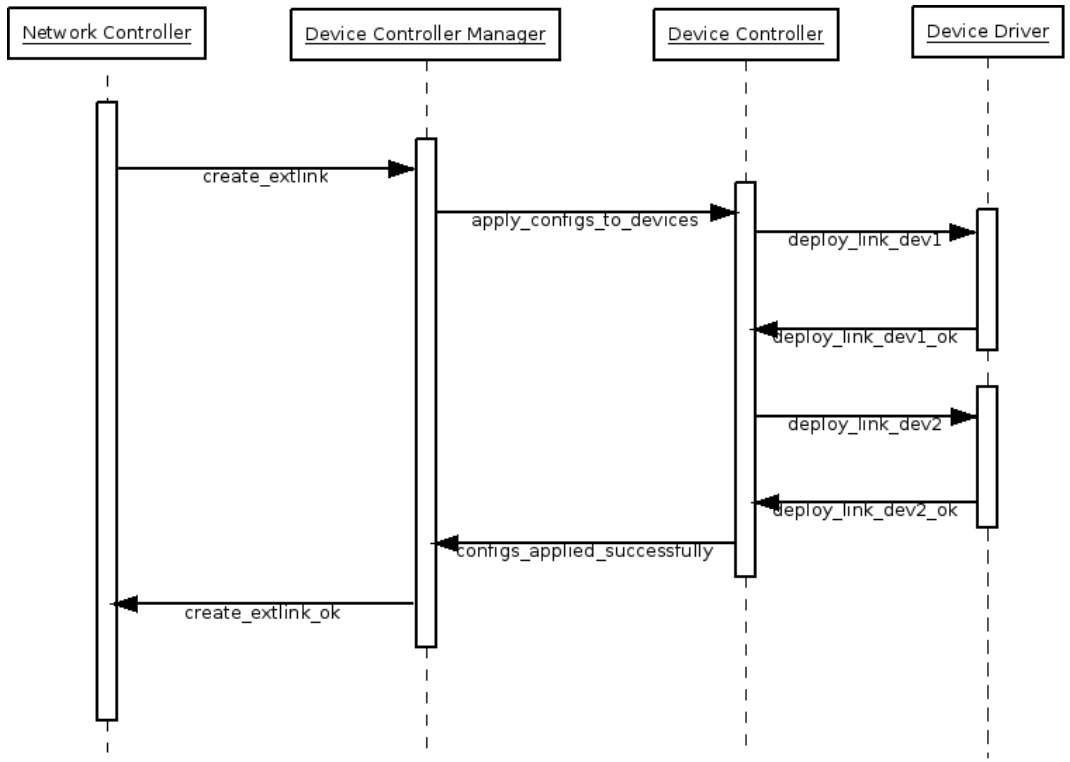

Figura 5.9: Diagrama de sequência representativo da troca de mensagens efetuada entre módulos da *framework* EXTNET

5.6 EXTNET FRAMEWORK - NEUTRON - CARACTERÍSTICAS DOS *paths* CONSTRUÍDOS NA REDE EXTERNA

Os *path* criados na rede externa podem ser constituídos por *ExtLinks* baseados em túneis *overlay* ou em VLAN, consoante as interfaces nos *endpoints*. Neste caso sendo tendo a implementação como *target* o uso de dispositivos Cisco com o OS IOS, na interceção dos *ExtLinks* nos dispositivos, optou-se por usar a capacidade de criar *bridges* estas são chamadas de `bridge-groups` no IOS. Com o uso deste recurso, é possível criar *bridges* L2 de forma a colocar, interfaces físicas e virtuais dos dispositivos no

mesmo domínio de colisão. Quando é necessário interligar duas ou mais interfaces num contexto de uma rede virtual específica, é criada uma *bridge* associada a essa rede virtual, onde são colocadas as interfaces e os *endpoints* dos túneis relativas a essa rede. A associação com a rede virtual, é mantida no ficheiro de configuração JSON associado ao dispositivo em questão para o contexto dessas configurações específicas do dispositivo não serem perdidas. Um exemplo de um destes ficheiros encontra-se no apêndice B. Os pares *network_id bridge_group_id* são guardados no ficheiro JSON no atributo de nome `bridge_groups_attributed`.
Importa ainda referir de que forma são geridos os VLAN IDs, no caso de segmentos que só suportam VLAN. Aquando da recolha de informação acerca da topologia, a *framework* recolhe os VLAN IDs disponíveis em cada dispositivo. Em cada segmento (`ExtSegment`), é verificada a disponibilidade em comum entre os dispositivos que integram o segmento (`ExtSegment`), os VLAN IDs em comum são guardados no `ExtSegment` correspondente no atributo `segmentation_id` na base de dados. No caso dos `ExtSegments` que só suportam túneis GRE, estes possuem um conjunto pré-definido IDs na configuração. A atribuição dos IDs ao `ExtLinks` é feita de forma automática tirando e repondo os IDs, consoante as operações de criação e a remoção dos `ExtLinks`. Entre dispositivos com interfaces em segmentos diferentes, a "ponte" entre as VLANs e os túneis GRE é feita usando as já mencionadas *bridges* L2, desta forma consegue-se um melhor aproveitamento dos IDs disponíveis nos dispositivos.
Importa ainda referir, de que forma são tratados os *loops* nas extensões criadas pela *framework*. Nesta implementação, devido ao uso dos `bridge-groups` do Cisco IOS, é delegada a responsabilidade de resolver os *loops* nas *bridge-groups* através da implementação do protocolo *Spanning Tree Protocol* (STP) nela existente. A ativação do protocolo STP nas `bridge-groups` é da responsabilidade do *Device Driver*.

5.7 EXTNET FRAMEWORK - NEUTRON - SETUP E PARÂMETROS DE CONFIGURAÇÃO

Todo o *setup* usado é baseado no projeto de desenvolvimento do OpenStack de nome DevStack, as configurações da *framework* EXTNET são efetuadas através das extensão do ficheiro de configuração principal do Neutron, que normalmente reside na pasta `/etc/neutron` com o nome de `neutron.conf`. Neste ficheiro, para a *framework* conseguir aplicar as configurações, terão de constar os `Device Controller` disponíveis e o *hostname* dos dispositivos a cada um deles associados, esta opção tem o nome de `device_controllers`. É necessário também incluir neste ficheiro, informações (IP, *Netmask* e *hostname*) acerca do dispositivo *next-hop*. Estas informações vão ser necessárias, para que a *framework* possa saber por onde começar a descoberta da topologia quando necessário. Foram também incluídas algumas informações acerca do OVS que está correr no *Controller Node*, por forma a criar o respetivo `ExtNode` no modelo de dados da *framework*. O script Python que permite a adição destas configurações no ficheiro `neutron.conf` está disponível no Apêndice A.
A *framework* pode ser instalada usando o mecanismo de instalação existente na linguagem Python, executando o seguinte comando dentro da pasta `python setup.py install`

capítulo 6

Avaliação da Solução

Para testar a framework, foi elaborado um cenário virtual que proporciona características idênticas às redes onde a framework seria utilizada, como por exemplo a rede existente na Universidade de Aveiro. Este cenário vai permitir analisar várias métricas, e com essa análise determinar se a framework, possui um grau de viabilidade suficiente para ser usada em ambiente real.

6.1 cenário de testes

O cenário de testes escolhido foi um ambiente virtualizado, constituído por um portátil Apple MacBook Pro Early 2015 cujas características principais são, um processador Intel Core i5 5257U 2.7GHz, 8GB LPDDR3 de memória RAM e 128GB de disco SSD com interface PCI Express. Esta máquina vai ser o *host*, onde vai estar correr o seguinte software que nos vai permitir, construir um cenário de rede minimamente idêntico ao *target* respetivo numa rede real. Em termos de software, foi escolhido o *Graphical Network Simulator-3* (GNS3)[1] como plataforma de virtualização da topologia, dando suporte ao uso de dispositivos cisco IOS de forma virtual. Como *hypervisor* para poder usar máquinas virtuais, que serão ligadas a essa topologia criada no GNS3, vai ser usado o Oracle VirtualBox. Além disso, o GNS3 possui mecanismos para interagir com VMs geridas pelo VirtualBox, promovendo assim uma melhor integração das VMs na topologia de testes a correr no GNS3.
Em termos de VMs, foi utilizado como *host* da plataforma DevStack uma VM baseada em Debian 8.3, a nível de recursos esta VM tem 4GB de RAM e 1 CPU. Como VMs para simular máquinas na rede externa, foi utilizado o Linux Tiny Core. Esta escolha deveu-se ao seu pequeno tamanho e aos baixos requisitos de RAM, em que assim é possível colocar várias máquinas deste tipo a correr em simultâneo sem constrangimentos ao nível de RAM. No *host* Debian que aloja o DevStack, as VMs usadas no ambiente OpenStack para teste da conectividade, são formadas a partir de imagens de CirrOS[2]. Este OS é baseado no Linux Tiny Core, a sua principal diferença para este, é que possui otimizações incluídas para correr em ambientes *Cloud*.
Como já foi mencionado nas secções anteriores, os dispositivos de rede escolhidos para testar a *framework* foram os Cisco EtherSwitch, isto pela facilidade com que se consegue emular este tipo de

[1] https://www.gns3.com/
[2] https://launchpad.net/cirros

hardware precisamente no GNS3. O GNS3 possui um suporte bastante adequado para gerir *hypervisors* que suportem o Cisco IOS que é um OS baseado na arquitetura MIPS. Neste caso para emular os Cisco EtherSwitch foi usado o Dynamips como *hypervisor*.
Como cenário de teste foi usado o seguinte ilustrado na figura 6.1.

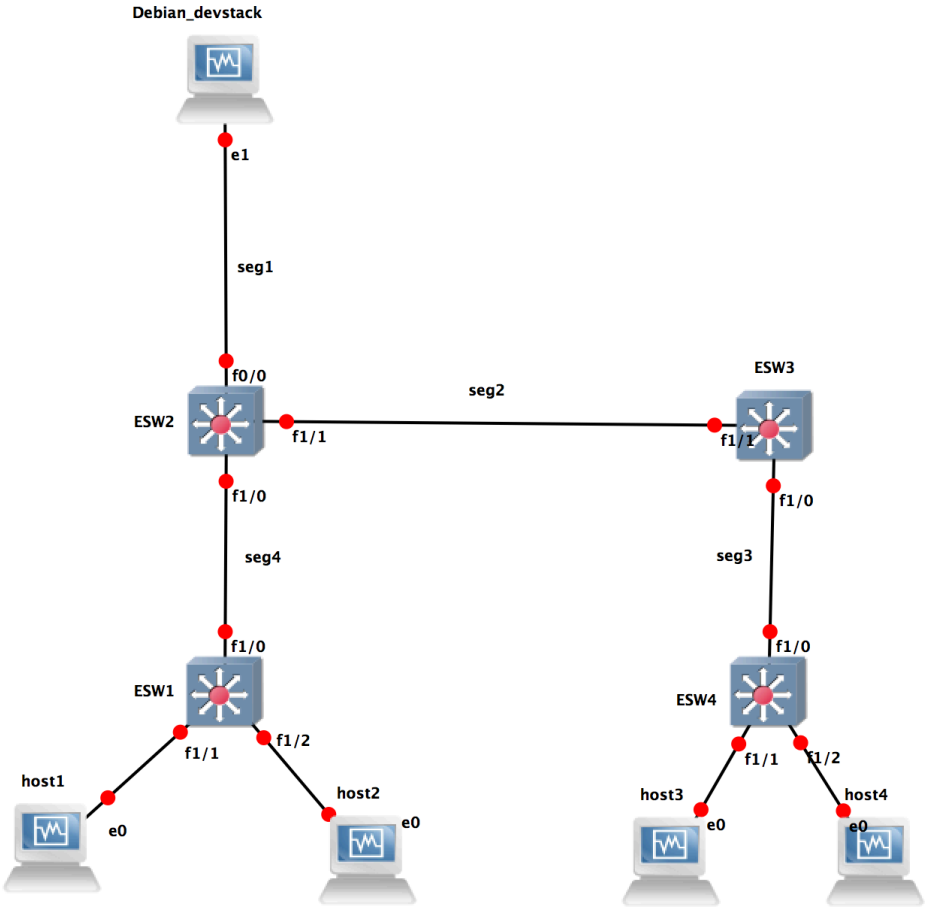

Figura 6.1: Cenário de testes da *framework* EXTNET

Pretende-se com este cenário recriar, de certa forma, uma organização idêntica à da rede da existente na Universidade de Aveiro, em que o *host* `Debian_devstack` representa a *Cloud* existente no *datacenter*, o switch `ESW2` representa o *gateway* do *datacenter*, o `ESW3` e o `ESW1` representam switches à entrada de dois departamentos. E por fim, o switch `ESW4` representa um switch relativo a uma sala de aula.

6.2 TESTES

Em relação aos testes efetuados, estes foram realizados com o Neutron configurado para usar túneis GRE como tecnologia *overlay*. Foram feitas medições de tempo do *setup* e do *teardown* da criação de uma porta, num total de 10 vezes cada. Estes testes não têm em conta a carga do *host* onde está a correr o GNS3 e a VM com o DevStack.

As medições de tempo foram realizadas usando as diferenças temporais, verificadas através do WireShark[3], entre o primeiro pacote relacionado com as configurações do primeiro *endpoint*, que neste caso é a bridge `br-int`, e o último pacote relativo às configurações do último *endpoint* que é o switch `ESW4`. Assim é possível medir efetivamente, o tempo que demora a aplicação das configurações nos diapositivos, sem ter os tempos de processamento da API que neste caso são irrelevantes. Um exemplo do comando usado na CLI do Neutron (`python-neutronclient`) para fazer o Teste 1 é o seguinte: `neutron port-create 74bdac4a-9864-4c08-bcc4-26f734730f24 --mac-address 08:00:27:4b:9a:cc --extnode-name ESW4 --extinterface-name FastEthernet1/1`.
Estes testes não têm em consideração o tempo de atribuição do endereço IP pelo servidor DHCP gerido pelo Neutron. Ou seja, não tem em consideração o tempo que leva até a ligação estar efetivamente operacional no dispositivo externo.

6.2.1 TESTE 1

Este teste consiste em fazer a medição temporal de quanto demora a operação de criar uma porta externa, neste caso a base de dados não possui ainda informação acerca da topologia da rede externa. Portanto, ao criar a porta externa a *framework* vai estar no pior caso possível, em que não possui nenhuma informação acerca da rede externa. Posto isto, ao fazer o pedido de criação da porta externa, a *framework* terá de fazer o mapa da topologia, calcular o *best path* e criar os `ExtLinks` para ligar a porta externa à rede gerida pelo OpenStack. Este teste cria os seguintes `ExtLinks`. No *1-hop* um túnel GRE, no *2-hop* e *3-hop* são criadas VLAN com diferentes VLAN IDs. Este teste vai permitir-nos simular um cenário idêntico, ao caso de uso da Universidade de Aveiro. Ou seja, vai criar uma ligação entre o *datacenter* e uma sala num determinado departamento, e verificar o impacto temporal do uso da *framework* no pior caso possível. Os links criados neste teste estão representados na figura 6.2.

[3]https://www.wireshark.org

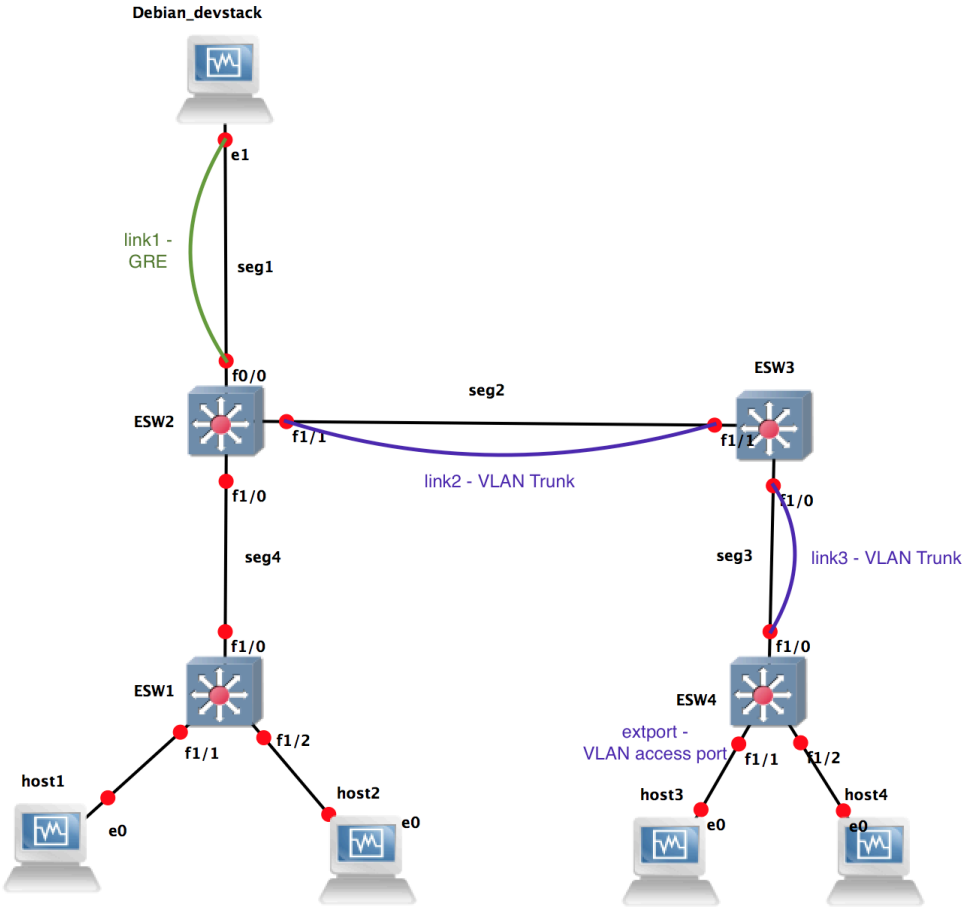

Figura 6.2: Links criados no teste 1 da *framework* EXTNET

RESULTADOS DO TESTE 1

Os resultados obtidos neste teste podem ser observados nas seguintes tabelas:

(a) Média e desvio padrão do *setup time*

Média	67.400
Desvio Padrão	0.957

(b) Média e desvio padrão do *teardown time*

Média	33.608
Desvio Padrão	0.841

Tabela 6.1: Médias e desvios padrão do teste 1

É possível observar nos resultados deste primeiro teste, que os valores obtidos rondam o minuto e 17 segundos para fazer o *setup*. E sensivelmente 34 segundos para fazer o *teardown*, de uma porta externa que está a *4-hop* de distância. Esta diferença poderá advir do facto, de o número de comandos a serem executados remotamente nos dispositivos ao fazer *teardown*, ser inferior em relação aos de *setup*. Consegue-se já verificar que, estes tempos medidos são altamente penalizados pelo tempo de inserção e assimilação dos comandos nos dispositivos de rede externos.
Normalmente, este tipo de cenário apresentado em que é necessário estender as redes virtuais para o exterior, é concretizado ainda pelo administrador de rede configurando cada dispositivo de forma individual. Ao comparar com os resultados da solução apresentada, o tempo despendido para a execução da tarefa pelo administrador é incomparavelmente superior.

6.2.2 TESTE 2

Este teste é idêntico ao anterior, apenas difere no facto de que quando é feito o pedido da criação da porta externa (`ExtPort`), a *framework* já possui a informação da topologia na base de dados. As operações que são feitas são o cálculo do *path*, a aplicação das configurações para obter os links (`ExtLinks`) que o compõem, e por fim a configuração da interface do dispositivo que aloja a porta externa. Os valores obtidos são mostrados a seguir.

RESULTADOS DO TESTE 2

(a) Média e desvio padrão do *setup time*

Média	52.321
Desvio Padrão	0.402

(b) Média e desvio padrão do *teardown time*

Média	33.730
Desvio Padrão	0.270

Tabela 6.2: Médias e desvios padrão do teste 2

Neste teste é possível observar que, o tempo que demora a recolha da informação acerca da topologia é cerca de de 22% do valor total do tempo que demora a porta a fazer *setup*. Continua-se a verificar aquilo que já se tinha notado, de certo modo, no teste 1. O tempo de *setup* é altamente influenciado pela operações de execução dos comandos nos dispositivos de rede. Este tempo é altamente imprevisível devido a várias condicionantes incontroláveis, como por exemplo, o congestionamento da rede, o tipo de protocolo em utilização (telnet, SSH, SNMP, ou outro), capacidades de processamento e características específicas dos dispositivos de rede. O tempo de recolha de informação acerca da topologia, é efetivamente significativo e igualmente altamente influenciável pelos fatores já enunciados. Acresce ainda o facto de que, devido à disparidade das características dos dispositivos, suporte do protocolo que está a ser usado para recolher essa informação, e mais uma vez, tendo em conta as condições da rede aquando da recolha (número de dispositivos ligados e outras mudanças na topologia), este valor pode mudar drasticamente de *deployment* para *deployment*.

CAPÍTULO 7

Conclusão

Neste capítulo é feita uma retrospetiva do trabalho realizado nesta dissertação, isto é, pretende-se avaliar o que foi proposto como trabalho a realizar e o que foi efetivamente efetuado. Pretende-se também neste capítulo abordar o trabalho futuro a ser realizado no âmbito desta solução.

7.1 Avaliação do trabalho efetuado

Em relação à avaliação sobre o trabalho efetuado, o objetivo de criar uma *framework* que fosse capaz de dotar um CMS de capacidades para estender as suas redes virtuais para fora do *datacenter*, foi efetivamente cumprido, mas com algumas reservas. Foi conseguido usando o CMS OpenStack que, as redes virtuais fossem estendidas de uma forma automática, sem a necessidade de o utilizador inserir as informações relativos à rede externa de forma manual. Embora o trabalho realizado careça de algumas funcionalidades que se podem considerar importantes, como a capacidade de monitorizar a topologia da rede externa e a capacidade de *rollback* no caso de haver erros na aplicação das configurações. Pode-se considerar que o objetivo de estender as redes virtuais geridas por um CMS, foi efetivamente cumprido.

Os maiores entraves neste trabalho prenderam-se com a questão da heterogeneidade dos dispositivos *legacy*. Conclui-se que para esse problema terá sempre de se encontrar uma solução de compromisso. Foi feita uma tentativa de abordagem nesse sentido, por exemplo com a criação dos *Device Drivers* e toda a sua gestão envolvente, em que foi necessário recorrer à escrita individual para cada tipo de modelo dos dispositivos de rede. Isto porque, mesmo tendo a solução a capacidade de recolher informações acerca dos dispositivos *legacy*, informações como as credenciais de acesso à linha de comandos obviamente que não são disponibilizadas, tendo portanto de haver outra forma de fornecer esta informação à *framework*. A forma encontrada, foi o uso de ficheiros de configuração que possuem todas as informações que não podem ser recolhidas automaticamente.

Em relação à recolha de informação dos dispositivos de rede, nesta dissertação apenas é possível fazer essa recolha somente a dispositivos da marca Cisco. Além disso devido ao facto de se estar a usar SNMP, e os fabricante implementarem as MIBs de forma diferente, é pouco provável que esta implementação consiga recolher informação de dispositivos de outras marcas, mas tal facto não foi efetivamente testado. Logo o facto de recolher apenas informações de dispositivos Cisco é uma

limitação deste trabalho. Um problema encontrado que compromete num certo ponto a implementação feita no OpenStack, é o facto de a solução estar a bloquear a *main thread* do `neutron-server`. Isto deve-se a esta ter de ficar à espera que a aplicação das configurações, durante a criação de uma porta externa termine. Como a aplicação das configurações é uma tarefa normalmente mais demorada, devido às incertezas acerca do congestionamento da rede, o tempo que o `neutron-server` fica sem atender pedidos pode ser considerável, o que invalida o uso da solução em muitos *deployments* do OpenStack. A solução possui ainda limitações em relação à linguagem de programação usada, isto deve-se ao facto de ao ser uma *framework* que fornece um conjunto de APIs Python. Devido a este facto, torna-se difícil usar a *framework* em CMSs que não tenham sido desenvolvidos nesta mesma linguagem de programação.
No entanto, os resultados obtidos em torno do objetivo principal, foram positivos. Os tempos que foram obtidos no *setup* e *teardown*, demonstram que a solução é viável num contexto mais alargado e com mais funcionalidades disponíveis.

7.2 TRABALHO FUTURO

No âmbito desta solução, existem múltiplas funcionalidades importantes a adicionar como trabalho futuro. As funcionalidades mais importantes a serem consideradas, são ao nível da gestão da informação acerca da topologia da rede externa acessível pela *framework*. Neste trabalho apenas se recolhe a informação da topologia quando o dispositivo pretendido não se encontra na base de dados, o que é uma opção facilmente falível, devido à possível mudança da topologia durante os possíveis pedidos de portas externas que levem a nova pesquisa. É, portanto, perentório desenvolver um mecanismo de monitorização do estado da topologia que, consiga atualizar a informação presente na base de dados de uma forma mais constante. Este mecanismo de monitorização, deverá ser capaz de adicionar e remover novos dispositivos da base de dados, assim que eles estejam disponíveis na topologia. Isto pode ser conseguido através de pesquisas à topologia periódicas, com intervalos de tempo reduzidos. Este possível mecanismo de monitorização envolveria alterações no *Network Controller*, de forma a responder com as ações necessárias às *callbacks* do *Device Controller Manager*, em que o *Device Controller Manager* seria informado pelos *Device Controllers* acerca estado dos dispositivos.
Outra funcionalidade importante a implementar seria um mecanismo de *rollback*, para que quando houvesse uma possível falha a meio de uma operação de configuração dos dispositivos de rede externos, fosse possível fazer o *rollback* das configurações já aplicadas nos dispositivos para o momento antes da falha ocorrer. Esta funcionalidade é muito importante para manter a consistência das configurações aplicadas nos dispositivos. Esta funcionalidade é também igualmente importante, no caso de um dos dispositivos de rede que suporta alguns dos *links* sofrer algum problema ou avaria. Neste caso, por exemplo, o sistema de monitorização poderia despoletar um alerta para o *Network Controller*, por forma a reconstruir o *path* por outros segmentos de forma a manter a conectividade. O mecanismo de monitorização poderia ser incluído no componente *Device Controller Manager*.
Como trabalho futuro, deverá ser considerada ainda a hipótese de fazer com que as operações de pesquisa da topologia e a aplicação das configurações nos dispositivos externos, não bloqueiem o `neutron-server`. Isto porque, nesta implementação da solução, o `neutron-server` necessita de ficar à espera que estas operações terminem de forma a dar *feedback* ao utilizador acerca do sucesso da criação da porta externa. Uma possível solução para este problema seria o uso de *eventlets* na implementação

destas operações. O *Eventlet* é uma biblioteca Python específica que implementa concorrência em implementações de operações relacionadas com redes. No entanto, esta solução levanta mais alguns problemas, aquando do uso de um sistema de *rollback* das operações, que vai necessitar de mecanismos mais avançados para a reposição das configurações já aplicadas.
Em suma, embora exista algumas limitações ao nível do tipo de dispositivos *legacy* suportados, e possíveis limitações na gestão dos *links* por falta de monitorização destes, cumpriu-se o objetivo principal deste trabalho, que era estender as redes virtuais de umCMS para uma rede *legacy*. Recomenda-se, como possível investigação a o teste da *framework* com múltiplos algoritmos de pesquisa e mapeamento da topologia, para incitar um possível uso em ambiente de produção.

Referências

[1] P. Mell e T. Grance, «The nist definition of cloud computing», *NIST Special Publication*, vol. 145, p. 7, 2011, ISSN: 00845612. DOI: 10.1136/emj.2010.096966. arXiv: 2305-0543. endereço: http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf.

[2] D. S. Dirk Krafzig, Karl Banke, *Enterprise soa: service-oriented architecture best practices*, 2004. endereço: http://entropy.soldierx.com/%7B~%7Dkayin/archive/ebooks/Prentice%20Hall%20-%20Enterprise%20SOA.%20Service-Oriented%20Architecture%20Best%20Practices.pdf.

[3] T. Erl, *Soa: principles of service design*. Prentice Hall, 2008, vol. 1, p. 573, ISBN: 9780132344821.

[4] N. M. M. K. Chowdhury e R. Boutaba, «A survey of network virtualization», *Computer Networks*, vol. 54, nº 5, pp. 862–876, 2010, ISSN: 13891286. DOI: 10.1016/j.comnet.2009.10.017.

[5] N. M. K. Chowdhury e R. Boutaba, «A survey of network virtualization», *Computer Networks*, vol. 54, nº 5, pp. 862–876, 2010.

[6] N. M. M. K. Chowdhury e R. Boutaba, «Network virtualization: State of the art and research challenges», *IEEE Communications Magazine*, vol. 47, nº 7, pp. 20–26, jul. de 2009, ISSN: 0163-6804. DOI: 10.1109/MCOM.2009.5183468.

[7] B. A. A. Nunes, M. Mendonca, X. N. Nguyen, K. Obraczka e T. Turletti, «A survey of software-defined networking: Past, present, and future of programmable networks», *IEEE Communications Surveys Tutorials*, vol. 16, nº 3, pp. 1617–1634, Third de 2014, ISSN: 1553-877X. DOI: 10.1109/SURV.2014.012214.00180.

[8] A. Wang, M. Iyer, R. Dutta, G. N. Rouskas e I. Baldine, «Network virtualization: Technologies, perspectives, and frontiers», *Journal of Lightwave Technology*, vol. 31, nº 4, pp. 523–537, fev. de 2013, ISSN: 0733-8724. DOI: 10.1109/JLT.2012.2213796.

[9] S. Azodolmolky, P. Wieder e R. Yahyapour, «Sdn-based cloud computing networking», em *2013 15th International Conference on Transparent Optical Networks (ICTON)*, jun. de 2013, pp. 1–4. DOI: 10.1109/ICTON.2013.6602678.

[10] Open Networking Foundation, «Software-defined networking: the new norm for networks [white paper]», *ONF White Paper*, pp. 1–12, 2012.

[11] K. Bakshi, «Network considerations for open source based clouds», em *IEEE Aerospace Conference Proceedings*, vol. 2015-June, 2015, ISBN: 9781479953790. DOI: 10.1109/AERO.2015.7118997.

[12] I. Stojmenovic e S. Wen, «The fog computing paradigm: scenarios and security issues», *Proceedings of the 2014 Federated Conference on Computer Science and Information Systems*, vol. 2, pp. 1–8, 2014, ISSN: 2300-5963. DOI: 10.15439/2014F503. endereço: https://fedcsis.org/proceedings/2014/drp/503.html.

[13] Cisco Systems, *Fog computing and the internet of things: extend the cloud to where the things are*, 2016. endereço: http://www.cisco.com/c/dam/en%7B%5C_%7Dus/solutions/trends/iot/docs/computing-overview.pdf (acedido em 24/08/2016).

[14] F. Bonomi, R. Milito, J. Zhu e S. Addepalli, «Fog computing and its role in the internet of things», em *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, sér. MCC '12, Helsinki, Finland: ACM, 2012, pp. 13–16, ISBN: 978-1-4503-1519-7. DOI: 10.1145/2342509.2342513. endereço: http://doi.acm.org/10.1145/2342509.2342513.

[15] N. Cook, D. Milojicic e V. Talwar, «Cloud management», *Journal of Internet Services and Applications*, vol. 3, pp. 67–75, 2012, ISSN: 1867-4828. DOI: 10.1007/s13174-011-0053-8.

[16] W. Voorsluys, J. Broberg e R. Buyya, «Introduction to cloud computing», *Cloud Computing: Principles and Paradigms*, pp. 1–41, 2011, ISSN: 1542-4065. DOI: 10.1002/9780470940105.ch1.

[17] *Chapter 1. architecture - openstack installation guide for ubuntu 14.04 - juno*. endereço: http://docs.openstack.org/juno/install-guide/install/apt/content/ch%7B%5C_%7Doverview.html (acedido em 28/09/2016).

[18] *Openstack style guidelines — hacking 0.11.1.dev18 documentation*. endereço: http://docs.openstack.org/developer/hacking/ (acedido em 21/09/2016).

[19] B. Davie, *Network virtualization gets physical*, 2013. endereço: https://cto.vmware.com/network-virtualization-gets-physical/ (acedido em 25/08/2016).

[20] D. Bruce e K. Duda, *Physical networks in the virtualized networking world - the network virtualization blog*, 2014. endereço: https://blogs.vmware.com/networkvirtualization/2014/07/physical-virtual-networking.html%7B%5C#%7D.V77yF5MrKRs (acedido em 25/08/2016).

[21] F. N. N. Farias, J. J. Salvatti, E. C. Cerqueira e A. J. G. Abelem, «A proposal management of the legacy network environment using openflow control plane», em *Proceedings of the 2012 IEEE Network Operations and Management Symposium, NOMS 2012*, 2012, pp. 1143–1150, ISBN: 9781467302685. DOI: 10.1109/NOMS.2012.6212041.

[22] F. Manco, *Network infrastructure control for virtual campus*, 2013. endereço: http://hdl.handle.net/10773/12725.

[23] *Neutron campus network extension : blueprints : neutron*. endereço: https://blueprints.launchpad.net/neutron/+spec/campus-network (acedido em 28/09/2016).

[24] I. D. Cardoso, *Network infrastructure control for virtual campuses*, 2014. endereço: http://hdl.handle.net/10773/14707.

[25] *Neutron l2 gateway + hp 5930 switch ovsdb integration, for vxlan bridging and routing*. endereço: http://kimizhang.com/neutron-l2-gateway-hp-5930-switch-ovsdb-integration/ (acedido em 28/09/2016).

[26] F. Bonomi, R. Milito, J. Zhu e S. Addepalli, «Fog computing and its role in the internet of things», *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pp. 13–16, 2012, ISSN: 978-1-4503-1519-7. DOI: 10.1145/2342509.2342513. endereço: http://doi.acm.org/10.1145/2342509.2342513$%5Cbackslash$npapers2://publication/doi/10.1145/2342509.2342513.

[27] *Openstack docs: networking architecture*. endereço: http://docs.openstack.org/security-guide/networking/architecture.html (acedido em 24/09/2016).

[28] R. Kukura e K. Mestery, *Modular layer 2 in openstack neutron*. endereço: http://www.slideshare.net/mestery/modular-layer-2-in-openstack-neutron (acedido em 24/09/2016).

[29] J. Denton, *Learning OpenStack networking (Neutron) architect and build a network infrastructure for your cloud using OpenStack Neutron networking*. Packt Pub, 2014, ISBN: 9781783983308.

[30] *Python patterns - implementing graphs | python.org*. endereço: `https://www.python.org/doc/essays/graphs/` (acedido em 30/09/2016).

APÊNDICE A

```
from oslo_config import cfg

netctrl_group = cfg.OptGroup(name='EXTNET_CONTROLLER',
                             title='Network controller default options.')
netctrl_opts = [
    cfg.StrOpt('name',
               default='OVS',
               required=True,
               help='Network controller name.'),

    cfg.StrOpt('ip_address',
               default='192.168.2.2',
               required=True,
               help='Network controller IP address.'),

    cfg.StrOpt('netmask',
               default='255.255.255.0',
               required=True,
               help='Network controller netmask.'),

    cfg.StrOpt('ids_available',
               default='0:10',
               required=True,
               help='Network IDs available on the first hop segment.'),

    cfg.StrOpt('nexthop_ip',
               default='192.168.2.1',
               required=True,
               help='Next hop device IP address for access by the network
                   controller.'),

    cfg.StrOpt('nexthop_name',
               default='ESW2',
               required=True,
               help='Next hop device name connected to network controller.'),

    cfg.DictOpt('nexthop_interface',
                default='name: FastEthernet0/0,ip_address: 192.168.2.1',
                required=True,
                help='Next hop device interface connected to network
                    controller.'),

    cfg.DictOpt('device_controllers',
                default="q-agent-notifier: OVS,extnet_agent:
                    ESW1;ESW2;ESW3;ESW4",
                required=True,
                help='Device controller available on the network controller.'),
]
cfg.CONF.register_group(netctrl_group)
cfg.CONF.register_opts(netctrl_opts, netctrl_group)
```

Listing 3: Opções adicionadas ao ficheiro “neutron.conf”

APÊNDICE B

```
{
  "bridge_groups_available": "22:200",
  "device_driver":
      "Cisco3700:/home/mfernandes/drivers_and_configs/cisco3700.py",
  "password": "pass",
  "bridge_groups_attributed": {"e26f0f6a-80b7-426d-95c0-1b7b2ac26385": 18},
  "port": 23
}
```

Listing 4: Exemplo de ficheiro JSON com informações acerca de um dispositivo. Este ficheiro é usado em conjunto com um device driver. Neste caso o device driver associado é o "Cisco3700".

APÊNDICE C

```
def get_devices_info(self, ip_address, **kwargs):
    self.mech_obj.connect(ip_address)
    node_info = self.mech_obj.get_node_info_dict()
    if node_info:
        node, node_dict = node_info.items()[0]
        self.nodes_info[node] = node_dict
        for interface in node_dict['interfaces']:
            n_hops_list = interface['next_hops']
            if n_hops_list:
                for ip_address in n_hops_list:
                    self.mech_obj.connect(ip_address)
                    node_name = self.mech_obj.get_node_name()
                    if node_name and node_name not in self.nodes_info.keys():
                        self.get_devices_info(ip_address)
    return self.nodes_info
```

Listing 5: Função recursiva “get_devices_info”

APÊNDICE D

Assinaturas das classes e respetivos métodos da framework EXTNET.

```
@six.add_metaclass(abc.ABCMeta)
class ExtNetNetworkController(object):

    @abc.abstractmethod
    def deploy_link(self, link,
                    link_type,
                    interface1,
                    interface2,
                    node1,
                    node2,
                    **kwargs):
        pass

    @abc.abstractmethod
    def undeploy_link(self, link,
                      link_type,
                      interface1,
                      interface2,
                      node1,
                      node2,
                      **kwargs):
        pass

    @abc.abstractmethod
    def deploy_port(self, interface, node, segmentation_id, **kwargs):
        pass

    @abc.abstractmethod
    def undeploy_port(self, interface, node, segmentation_id, **kwargs):
        pass
```

Listing 6: Assinaturas relativas à classe "Network Controller"

```
@six.add_metaclass(abc.ABCMeta)
class TopologyDicoveryApi(object):

    @abc.abstractmethod
    def get_devices_info(self, ip_address, **kwargs):
        pass


@six.add_metaclass(abc.ABCMeta)
class TopoDiscMechanismApi(object):

    @abc.abstractmethod
```

```
    def connect(self, hostname, **kwargs):
        pass

    @abc.abstractmethod
    def get_node_name(self):
        pass

    @abc.abstractmethod
    def get_node_interfaces_up(self):
        pass

    @abc.abstractmethod
    def get_node_info_dict(self):
        pass
```

Listing 7: Assinaturas relativas à classe ao componente "Topology Discovery"'

```
@six.add_metaclass(abc.ABCMeta)
class ExtNetNetworkMapper(object):

    @abc.abstractmethod
    def build_best_path(self, **kwargs):
        pass
```

Listing 8: Assinaturas relativas à classe "Network Mapper"

```
@six.add_metaclass(abc.ABCMeta)
class ExtNetDeviceControllerManager(object):

    @abc.abstractmethod
    def get_device_controller(self, **kwargs):
        pass

    @abc.abstractmethod
    def deploy_port_on_node(self, interface, node, segmentation_id, **kwargs):
        pass

    @abc.abstractmethod
    def undeploy_port_on_node(self, interface, node, segmentation_id,
        **kwargs):
        pass

    @abc.abstractmethod
    def deploy_link_on_node(self, interface, node, segmentation_id,
        network_type, **kwargs):
        pass
```

```
    @abc.abstractmethod
    def undeploy_link_on_node(self, interface, node, segmentation_id,
        network_type, **kwargs):
        pass
```

Listing 9: Assinaturas relativas à classe "Device Controller Manager"

```
@six.add_metaclass(abc.ABCMeta)
class ExtNetDeviceController(object):

    @abc.abstractproperty
    def device_controller_name(self):
        pass

    @abc.abstractproperty
    def get_managed_devices(self):
        pass

    @abc.abstractmethod
    def deploy_link(self, ctxt, interface, node, network_type,
        segmentation_id, **kwargs):
        pass

    @abc.abstractmethod
    def undeploy_link(self, ctxt, interface, node, network_type,
        segmentation_id, **kwargs):
        pass

    @abc.abstractmethod
    def deploy_port(self, ctxt, interface, node, segmentation_id, **kwargs):
        pass

    @abc.abstractmethod
    def deploy_port(self, ctxt, interface, node, **kwargs):
        pass

    @abc.abstractmethod
    def load_driver(self, device_name, device_driver):
        pass
```

Listing 10: Assinaturas relativas à classe "Device Controller"

```
@six.add_metaclass(abc.ABCMeta)
class ExtNetDeviceDriverBase(object):

    def __init__(self, device_name, ip_address, dev_config_location):
        self.device_name = device_name
```

```
        self.ip_address = ip_address
        self.dev_config_location = dev_config_location
        self.dev_config_dict = self.load_device_configs()

    def load_device_configs(self):
        with open(os.path.join(self.dev_config_location, self.device_name +
            '.json')) as device_json:
            config_dict = json.load(device_json)
        return config_dict

    def save_device_configs(self):
        with open(os.path.join(self.dev_config_location, self.device_name +
            '.json'), 'w') as device_json:
            json.dump(self.dev_config_dict, device_json)

    @abc.abstractproperty
    def driver_protocol(self):
        pass

    @abc.abstractproperty
    def driver_name(self):
        pass

    @abc.abstractproperty
    def driver_overlay_types(self):
        pass

    @abc.abstractmethod
    def deploy_link(self,
                    link_type,
                    interface_name,
                    tun_destination,
                    segmentation_id,
                    **kwargs):
        pass

    @abc.abstractmethod
    def undeploy_link(self,
                      link_type,
                      interface_name,
                      segmentation_id,
                      **kwargs):
        pass

    @abc.abstractmethod
    def deploy_port(self, interface_type, interface_name,
        link_segmentation_id, **kwargs):
        pass

    @abc.abstractmethod
    def undeploy_port(self, interface_type, interface_name,
        link_segmentation_id, **kwargs):
        pass
```

Listing 11: Assinaturas relativas à classe “Device Driver”

APÊNDICE E

Atributo	Tipo	Atributo Necessário	CRUD	Valor por defeito	Validações	Descrição
extinterface_name	String	Não	CRU	Nenhum	Nenhuma	Nome da `ExtInterface` onde a `ExtPort` vai ser ligada.
extnode_name	String	Não	CRU	Nenhum	Nenhuma	Nome do `ExtNode` onde a `ExtPort` vai estar localizada.

Tabela 1: Atributos da API relativos à `ExtPort` (Apenas são mencionados os atributos adicionados como extensão à entidade *core* do Neutron chamada `Port`)

Atributo	Tipo	Atributo Necessário	CRUD	Valor por defeito	Validações	Descrição
id	String (UUID)	Sim	R	Gerado automaticamente	Não aplicável	UUID que identifica um `ExtLink`.
name	String	Não	CRU	no_name	Validação do tipo String	Nome dado à `ExtLink`.
segmentation_id	String	Não	R	Atribuído pela framework	Não aplicável	ID de segmentação atribuído ao *ExtLink*.
extinterface1_id	Não	String (UUID)	CR	Não aplicável	Validação do UUID	`ExtInterface` referente ao primeiro *endpoint* do `ExtLink`.
extinterface2_id	Não	String (UUID)	CR	Não aplicável	Validação do UUID	`ExtInterface` referente ao segundo *endpoint* do `ExtLink`.
network_id	Sim	String (UUID)	CRU	Não aplicável	Validação do UUID	ID da rede à qual este `ExtLink` é associado.
tenant_id	String (UUID)	Sim	CR	Não aplicável	Validação do UUID	ID do utilizador ao qual este `ExtLink` é associado.

Tabela 2: Atributos da API relativos ao `ExtLink`

Atributo	Tipo	Atributo Necessário	CRUD	Valor por defeito	Validações	Descrição
id	String (UUID)	Sim	R	Gerado automaticamente	Não aplicável	UUID que identifica um `ExtNode`.
name	String	Não	CRU	no_name	Validação do tipo String	Nome dado à `ExtInterface`.
type	String	Sim	CR	Nenhum	Nenhum	Tipo de rede suportado pela `ExtInterface`. ("l2" ou "l3")
ip_address	String	Não	CRU	Nenhum	Validação do endereço IP	Endereço IP atribuído à interface no caso de ser "l3".
extnode_id	String (UUID)	Sim	CR	Não aplicável	Validação do UUID	`ExtNode` onde a `ExtInterface` está integrada.
extsegment_id	String (UUID)	Sim	CRU	Não aplicável	Validação do UUID	`ExtSegment` onde a `ExtInterface` está ligada.
tenant_id	String (UUID)	Sim	CR	Não aplicável	Validação do UUID	ID do utilizador ao qual esta `ExtInterface` é associada.

Tabela 3: Atributos da API relativos à `ExtInterface`

Atributo	Tipo	Atributo Necessário	CRUD	Valor por defeito	Validações	Descrição
id	String (UUID)	Sim	R	Gerado automaticamente	Não aplicável	UUID que identifica um `ExtNode`.
name	String	Não	CRU	no_name	Validação do tipo String	Nome dado ao `ExtNode`.
ip_address	String	Não	CRU	Nenhum	Validação do endereço IP	Endereço IP para administração do *ExtNode*.
tenant_id	String (UUID)	Sim	CR	Não aplicável	Validação do UUID	ID do utilizador ao qual este `ExtNode` é associado.

Tabela 4: Atributos da API relativos ao `ExtNode`